Be
TM
OS

The Be Book Table of Contents

Kits and Servers

- The Application Kit
- The Device Kit
- The Game Kit
- The Interface Kit
- The Kernel Kit
- The Mail Kit
- The Media Kit
- The Midi Kit
- The Network Kit
- The OpenGL Kit
- The Storage Kit
- The Support Kit
- The Translation Kit

Special Topics

- Release Notes
- The Input Server
- Drag and Drop
- Drivers
- Tracker
- Deskbar
- Screen Saver
- Background Images

Appendices

- System Messages
- Keyboard Information

The Be Book,
...in lovely HTML...
for BeOS Release 5

Copyright © 2000 Be, Inc. All rights reserved.

The Application Kit Table of Contents

- The Application Kit
- Messaging
- Scripting
- BApplication
- BClipboard
- BCursor
- BHandler
- BInvoker
- BLooper
- BMessage
- BMessageFilter
- BMessageQueue
- BMessageRunner
- BMessenger
- BPropertyInfo
- BRoster
- Global Variables, Constants, and Defined Types

The Be Book **The Application Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Application Kit: Master Index

=

=	BMessageFilter
=	BMessage
=	BMessenger
==	BMessenger

A

AboutRequested()	BApplication
B_ABOUT_REQUESTED	Global Variables, Constants, and Defined Types
ActivateApp()	BRoster
AddBool()	BMessage
AddCommonFilterList()	BLooper
AddData()	BMessage
AddDouble()	BMessage
AddFilter()	BHandler
AddFlat()	BMessage
AddFloat()	BMessage
AddHandler()	BLooper
AddInt16()	BMessage

AddInt32()	BMessage
AddInt64()	BMessage
AddInt8()	BMessage
AddMessage()	BMessageQueue
AddMessage()	BMessage
AddMessenger()	BMessage
AddPoint()	BMessage
AddPointer()	BMessage
AddRect()	BMessage
AddRef()	BMessage
AddSpecifier()	BMessage
AddString()	BMessage
AddToRecentDocuments()	BRoster
AddToRecentFolders()	BRoster
Allocation	BLooper
AllowsTypeCode()	BPropertyInfo
B_ANY_DELIVERY	Global Variables, Constants, and Defined Types
B_ANY_SOURCE	Global Variables, Constants, and Defined Types
AppActivated()	BApplication
AppResources()	BApplication
B_APP_ACTIVATED	Global Variables, Constants, and Defined Types
app_info	Global Variables, Constants, and Defined Types

BApplication	BApplication
BApplication()	BApplication
~BApplication()	BApplication
Application Flags	Global Variables, Constants, and Defined Types
The Application Kit	The Application Kit
The Application Kit	The Application Kit
Application Messages	BApplication
Application Messages	Global Variables, Constants, and Defined Types
Archive()	BHandler
Archived Fields	BApplication
Archived Fields	BHandler
B_ARCHIVED_OBJECT	Global Variables, Constants, and Defined Types
ArgvReceived()	BApplication
B_ARGV_ONLY	Global Variables, Constants, and Defined Types
B_ARGV_RECEIVED	Global Variables, Constants, and Defined Types

---

B

The BHandler Class	Messaging
The BLooper Class	Messaging
The BMessage Class	Messaging
The BMessenger Class	Messaging

B_BACKGROUND_APP	Global Variables, Constants, and Defined Types
Basics	Scripting
be_app and Subclassing BApplication	BApplication
be_app	BApplication
be_app	Global Variables, Constants, and Defined Types
be_app_messenger	BApplication
be_app_messenger	Global Variables, Constants, and Defined Types
be_clipboard	Global Variables, Constants, and Defined Types
be_roster	Global Variables, Constants, and Defined Types
BeginInvokeNotify()	BInvoker
Broadcast()	BRoster

C

B_CANCEL	Global Variables, Constants, and Defined Types
B_COPY	Global Variables, Constants, and Defined Types
B_CUT	Global Variables, Constants, and Defined Types
Command()	BInvoker
Command()	BMessageFilter

Commands	Scripting
CommonFilterList()	BLooper
Constants and Defined Types	BMessageFilter
Constants	BLooper
Constants	Global Variables, Constants, and Defined Types
Constructing the Object and Running the Message Loop	BApplication
Constructor and Destructor	BApplication
Constructor and Destructor	BCursor
Constructor and Destructor	BHandler
Constructor and Destructor	BInvoker
Constructor and Destructor	BLooper
Constructor and Destructor	BMessageFilter
Constructor and Destructor	BMessageQueue
Constructor and Destructor	BMessageRunner
Constructor and Destructor	BMessage
Constructor and Destructor	BMessenger
Constructor and Destructor	BPropertyInfo
Constructor and Destructor	BRoster
CountHandlers()	BLooper
CountLockRequests()	BLooper
CountLocks()	BLooper
CountMessages()	BMessageQueue
CountNames()	BMessage
CountWindows()	BApplication

B_COUNT_PROPERTIES	Scripting
B_CREATE_PROPERTY	Global Variables, Constants, and Defined Types
B_CREATE_PROPERTY	Scripting
Creating and Sending Scripting Messages	Scripting
CurrentMessage()	BLooper
BCursor	BCursor
BCursor()	BCursor
~BCursor()	BCursor
Cursor Constants	Global Variables, Constants, and Defined Types
Cursor Data Format	BCursor
Cursor Data Format	BCursor

D

Data Members	BMessage
Defined Types	Global Variables, Constants, and Defined Types
delete	BMessage
B_DELETE_PROPERTY	Global Variables, Constants, and Defined Types
B_DELETE_PROPERTY	Scripting
DetachCurrentMessage()	BLooper
B_DIRECT_SPECIFIER	Global Variables, Constants, and Defined Types

B_DIRECT_SPECIFIER	Scripting
DispatchMessage()	BLooper
B_DISPATCH_MESSAGE	Global Variables, Constants, and Defined Types
DropPoint()	BMessage
B_DROPPED_DELIVERY	Global Variables, Constants, and Defined Types

E

EndInvokeNotify()	BInvoker
Example	Scripting
Examples	BHandler
B_EXCLUSIVE_LAUNCH	Global Variables, Constants, and Defined Types
B_EXECUTE_PROPERTY	Scripting

F

Features of the Fundamental Classes	Messaging
Filter()	BMessageFilter
FilterList()	BHandler
filter_hook	BMessageFilter
filter_result	BMessageFilter
filter_result Constants	Global Variables, Constants, and Defined Types

filter_result	Global Variables, Constants, and Defined Types
Filtering	BHandler
FiltersAnyCommand()	BMessageFilter
FindApp()	BRoster
FindBool()	BMessage
FindData()	BMessage
FindDouble()	BMessage
FindFlat()	BMessage
FindFloat()	BMessage
FindInt16()	BMessage
FindInt32()	BMessage
FindInt64()	BMessage
FindMatch()	BPropertyInfo
FindMessage()	BMessageQueue
FindMessage()	BMessage
FindMessenger()	BMessage
FindPoint()	BMessage
FindPointer()	BMessage
FindRect()	BMessage
FindRef()	BMessage
FindString()	BMessage
Finding a Function	Messaging
Finding a Handler	Messaging
FintInt8()	BMessage
flags	BRoster

Flatten()	BMessage
Flatten	BPropertyInfo
FlattenedSize()	BMessage
FlattenedSize()	BPropertyInfo
From Looper to Handler	Messaging

G

GetActiveAppInfo()	BRoster
GetAppInfo()	BApplication
GetAppInfo()	BRoster
GetAppList()	BRoster
GetCurrentSpecifier()	BMessage
GetInfo()	BMessageRunner
GetInfo()	BMessage
GetRecentApps()	BRoster
GetRecentDocuments()	BRoster
GetRecentFolders()	BRoster
GetRunningAppInfo()	BRoster
GetSupportedSuites()	BApplication
GetSupportedSuites()	BHandler
B_GET_PROPERTY	Global Variables, Constants, and Defined Types
B_GET_PROPERTY	Scripting
B_GET_SUPPORTED_SUITES	Global Variables, Constants, and Defined Types

Global Variables	BApplication
Global Variables	Global Variables, Constants, and Defined Types
Global Variables, Constants, and Defined Types	Global Variables, Constants, and Defined Types
Global Variables, Constants, and Defined Types	Global Variables, Constants, and Defined Types

H

Handler Associations	Messaging
HandlerAt()	BLooper
BHandler	BHandler
BHandler()	BHandler
~BHandler()	BHandler
HandlerForReply()	BInvoker
The Handler List	BHandler
Handling a Reply	Messaging
HasSpecifiers()	BMessage
HideCursor()	BApplication
Hook Functions	BApplication
Hook Functions	BHandler
Hook Functions	BLooper
Hook Functions	BMessageFilter

I

B_ID_SPECIFIER	Global Variables, Constants, and Defined Types
B_ID_SPECIFIER	Scripting
IndexOf()	BLooper
B_INDEX_SPECIFIER	Global Variables, Constants, and Defined Types
B_INDEX_SPECIFIER	Scripting
Inheritance and the Handler Chain	Messaging
InitCheck()	BMessageRunner
Instantiate()	BCursor
"InternalName"	BHandler
Invoke()	BInvoker
InvokeKind()	BInvoker
InvokeNotify()	BInvoker
BInvoker	BInvoker
BInvoker()	BInvoker
~BInvoker()	BInvoker
IsCursorHidden()	BApplication
IsEmpty()	BMessageQueue
IsEmpty()	BMessage
IsFixedSize()	BPropertyInfo
IsLaunching()	BApplication
IsLocked()	BLooper
IsReply()	BMessage
IsRunning()	BRoster

IsSourceRemote()	BMessage
IsSourceWaiting()	BMessage
IsSystem()	BMessage
IsTargetLocal()	BInvoker
IsTargetLocal()	BMessenger
IsValid()	BMessenger

L

Launch()	BRoster
Launch Constants	Global Variables, Constants, and Defined Types
B_LAUNCH_MASK	Global Variables, Constants, and Defined Types
B_LOCAL_SOURCE	Global Variables, Constants, and Defined Types
Lock()	BLooper
Lock()	BMessageQueue
LockLooper()	BHandler
LockLooperWithTimeout()	BHandler
LockTarget()	BMessenger
LockTargetWithTimeout()	BMessenger
LockWithTimeout()	BLooper
Locking	BLooper

LockingThread()	BLooper
Looper()	BHandler
BLooper	BLooper
BLooper()	BLooper
~BLooper()	BLooper
Looper()	BMessageFilter
LooperForThread()	BLooper
Looper Port Capacity	Global Variables, Constants, and Defined Types
B_LOOPER_PORT_DEFAULT_CAPACITY	BLooper
B_LOOPER_PORT_DEFAULT_CAPACITY	Global Variables, Constants, and Defined Types

---

M

MakeEmpty()	BMessage
Making Objects Scriptable	Scripting
Member Functions	BApplication
Member Functions	BHandler
Member Functions	BInvoker
Member Functions	BLooper
Member Functions	BMessageFilter
Member Functions	BMessageQueue
Member Functions	BMessageRunner
Member Functions	BMessage

Member Functions	BMessenger
Member Functions	BPropertyInfo
Member Functions	BRoster
Message()	BInvoker
BMessage	BMessage
BMessage()	BMessage
~BMessage()	BMessage
Message Constants	Global Variables, Constants, and Defined Types
MessageDelivery()	BMessageFilter
BMessageFilter	BMessageFilter
BMessageFilter()	BMessageFilter
~BMessageFilter()	BMessageFilter
Message Filters	Messaging
BMessage Ownership	BMessage
Message Protocols	Messaging
MessageQueue()	BLooper
BMessageQueue	BMessageQueue
BMessageQueue()	BMessageQueue
~BMessageQueue()	BMessageQueue
MessageReceived()	BHandler
MessageReceived()	BLooper
BMessageRunner	BMessageRunner
BMessageRunner()	BMessageRunner
~BMessageRunner()	BMessageRunner
MessageSource()	BMessageFilter

Message Specifiers	Global Variables, Constants, and Defined Types
message_delivery	BMessageFilter
message_delivery Constants	Global Variables, Constants, and Defined Types
message_delivery	Global Variables, Constants, and Defined Types
B_MESSAGE_NOT_UNDERSTOOD	Global Variables, Constants, and Defined Types
message_source	BMessageFilter
message_source Constants	Global Variables, Constants, and Defined Types
message_source	Global Variables, Constants, and Defined Types
Messages and Handlers	BLooper
Messaging	Messaging
Messaging	Messaging
"Messenger"	BHandler
Messenger()	BInvoker
BMessenger	BMessenger
BMessenger()	BMessenger
~BMessenger()	BMessenger
B_MIME_DATA	Global Variables, Constants, and Defined Types
B_MULTIPLE_LAUNCH	Global Variables, Constants, and Defined Types

N

"Name"	BApplication
Name()	BHandler
B_NAME_SPECIFIER	Global Variables, Constants, and Defined Types
B_NAME_SPECIFIER	Scripting
new	BMessage
NextHandler()	BHandler
NextMessage()	BMessageQueue
B_NO_REPLY	Global Variables, Constants, and Defined Types
B_NO_SPECIFIER	Global Variables, Constants, and Defined Types
B_NODE_MONITOR	Global Variables, Constants, and Defined Types
Notifiers and Observers	BHandler

O

ObscureCursor()	BApplication
Operators	BMessageFilter
Operators	BMessage
Operators	BMessenger
Other Topics	BApplication

P

B_PASTE	Global Variables, Constants, and Defined Types
B_PULSE	Global Variables, Constants, and Defined Types
PopSpecifier()	BMessage
port	BRoster
Port Capacity	BLooper
PostMessage()	BLooper
The PostMessage() Function	Messaging
PreferredHandler()	BLooper
Preparatory Reading	BMessage
Previous()	BMessage
PrintToStream()	BMessage
PrintToStream()	BPropertyInfo
Priority	BLooper
B_PROGRAMMED_DELIVERY	Global Variables, Constants, and Defined Types
Properties and Specifiers	Scripting
BPropertyInfo	BPropertyInfo
PropertyInfo()	BPropertyInfo
~BPropertyInfo()	BPropertyInfo
property_info	BPropertyInfo
Pulse()	BApplication

Q

B_QUERY_UPDATE	Global Variables, Constants, and Defined Types
Quit()	BApplication
Quit()	BLooper
QuitRequested()	BApplication
QuitRequested()	BLooper
B_QUIT_REQUESTED	Global Variables, Constants, and Defined Types

---

R

B_REPLY	Global Variables, Constants, and Defined Types
B_RANGE_SPECIFIER	Global Variables, Constants, and Defined Types
B_RANGE_SPECIFIER	Scripting
ReadyToRun()	BApplication
B_READY_TO_RUN	Global Variables, Constants, and Defined Types
Receiving a Message	Messaging
ref	BRoster
RefsReceived()	BApplication
B_REFS_RECEIVED	Global Variables, Constants, and Defined Types
B_REMOTE_SOURCE	Global Variables, Constants, and Defined Types

RemoveCommonFilterList()	BLooper
RemoveData()	BMessage
RemoveFilter()	BHandler
RemoveHandler()	BLooper
RemoveMessage()	BMessageQueue
RemoveName()	BMessage
ReplaceBool()	BMessage
ReplaceData()	BMessage
ReplaceDouble()	BMessage
ReplaceFlat()	BMessage
ReplaceFloat()	BMessage
ReplaceInt16()	BMessage
ReplaceInt32()	BMessage
ReplaceInt64()	BMessage
ReplaceInt8()	BMessage
ReplaceMessage()	BMessage
ReplaceMessenger()	BMessage
ReplacePoint()	BMessage
ReplacePointer()	BMessage
ReplaceRect()	BMessage
ReplaceRef()	BMessage
ReplaceString()	BMessage
Replies	Scripting
B_RESET_STATUS_BAR	Global Variables, Constants, and Defined Types
ResolveSpecifier()	BHandler

ReturnAddress()	BMessage
B_REVERSE_INDEX_SPECIFIER	Global Variables, Constants, and Defined Types
B_REVERSE_INDEX_SPECIFIER	Scripting
B_REVERSE_RANGE_SPECIFIER	Global Variables, Constants, and Defined Types
B_REVERSE_RANGE_SPECIFIER	Scripting
BRoster	BRoster
BRoster()	BRoster
~BRoster()	BRoster
Run()	BApplication
Run()	BLooper

S

B_SAVE_REQUESTED	Global Variables, Constants, and Defined Types
Scripting	Scripting
Scripting	Scripting
Scripting Suites and Properties	BApplication
Scripting Suites and Properties	BHandler
see	BHandler
B_SELECT_ALL	Global Variables, Constants, and Defined Types
Sem()	BLooper

SendMessage()	BMessenger
The SendMessage() Function	Messaging
SendReply()	BMessage
Sending a Message	Messaging
SetCommonFilterList()	BLooper
SetCount()	BMessageRunner
SetCursor()	BApplication
SetFilterList()	BHandler
SetHandlerForReply()	BInvoker
SetInterval()	BMessageRunner
SetMessage()	BInvoker
SetName()	BHandler
SetNextHandler()	BHandler
SetPreferredHandler()	BLooper
SetPulseRate()	BApplication
SetTarget()	BInvoker
SetTimeout()	BInvoker
B_SET_PROPERTY	Global Variables, Constants, and Defined Types
B_SET_PROPERTY	Scripting
ShowCursor()	BApplication
signature	BRoster
B_SIMPLE_DATA	Global Variables, Constants, and Defined Types
B_SINGLE_LAUNCH	Global Variables, Constants, and Defined Types

B_SKIP_MESSAGE	Global Variables, Constants, and Defined Types
The Specifier Stack	Scripting
B_SPECIFIERS_END	Global Variables, Constants, and Defined Types
StartWatchingAll()	BHandler
StartWatching()	BHandler
StartWatching()	BRoster
Static Functions	BApplication
Static Functions	BCursor
Static Functions	BHandler
Static Functions	BLooper
StopWatchingAll()	BHandler
StopWatching()	BHandler
StopWatching()	BRoster
"Suites"	BHandler
Suites	Scripting

T

Target()	BInvoker
Target()	BMessenger
Targets	BHandler
Team()	BLooper
Team()	BMessenger
team	BRoster

TeamFor()	BRoster
Thread()	BLooper
Timeout()	BInvoker
TypeCode()	BPropertyInfo
Types of Functions	BMessage

U

Unflatten()	BMessage
Unflatten()	BPropertyInfo
Unlock()	BLooper
Unlock()	BMessageQueue
UnlockLooper()	BHandler
B_UPDATE_STATUS_BAR	Global Variables, Constants, and Defined Types

---

W

WasDelivered()	BMessage
WasDropped()	BMessage
WindowAt()	BApplication
"Window"	BApplication

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageFilter

Derived from: *none*
Declared in: be/app/MessageFilter.h
Library: libbe.so
Allocation: Constructor only
more...

A BMessageFilter is a message-screening function that you "attach" to a BLooper or BHandler. The message filter sees messages just before they're dispatched (i.e. just before **`BLooper::DispatchMessage()`**), and can modify or reject the message, change the message's designated handler, or whatever else it wants to do—the implementation of the filter function isn't restricted.

To define a message filter, you have to provide a message-filtering function. You do this by implementing the **`Filter()`** hook function in a BMessageFilter subclass, or by supplying a **`filter_hook`** function to the BMessageFilter constructor. Only one filter function per object is called. If you implement **`Filter()`** *and* provide a **`filter_hook`** function, the **`filter_hook`** will win.

To attach a message filter to a looper, call **`BLooper::AddCommonFilter()`**. To add it to a handler, call **`BHandler::AddFilter()`**. Looper filters see all incoming messages; handler filters see only those messages that are targetted for that particular handler.

A BLooper or BHandler can have more than one message filter. Furthermore, a looper can have two sets of filters: a looper set and a handler set (keep in mind that BLooper is derived from BHandler). Looper filters are applied before handler filters.

A BMessageFilter object can be assigned to only one BHandler or BLooper at a time.

The BMessageFilter class is intended to be used as part of the system-defined messaging system. If you try to use one outside this system, your results may not be what you expect.

Hook Functions

`Filter()`

Constructor and Destructor

BMessageFilter()

```
BMessageFilter(message_delivery delivery,
    message_source source,
    uint32 command,
    filter_hook filter = NULL)
BMessageFilter(message_delivery delivery,
    message_source source,
    filter_hook filter = NULL)
BMessageFilter(uint32 command,
    filter_hook filter = NULL)
BMessageFilter(const BMessageFilter &object)
BMessageFilter(const BMessageFilter *object)
```

Creates and returns a new BMessageFilter. The first three arguments define the types of messages that the object wants to see:

- *delivery* specifies how the message must arrive: drag-and-drop (`B_DROPPED_DELIVERY`), programmatically (`B_PROGRAMMED_DELIVERY`), or either (`B_ANY_DELIVERY`). The default is `B_ANY_DELIVERY`.
- *source* specifes whether the sender of the message must be local vis-a-vis this app (`B_LOCAL_SOURCE`), remote (`B_REMOTE_SOURCE`), or either (`B_ANY_SOURCE`). The default is `B_ANY_SOURCE`.
- *command* is a command constant. If supplied, the **what** value of the incoming message must match this value.

Messages that don't fit the definition won't be sent to the object's filter function.

The *filter* argument is a pointer to a `filter_hook` function. This is the function that's invoked when a message needs to be examined (see `filter_hook` for the protocol). You don't have to supply a `filter_hook` function; instead, you can implement BMessageFilter's `Filter()` function in a subclass.

For more information, refer to the description of the member `Filter()` function.

~BMessageFilter()

```
virtual ~BMessageFilter()
```

Does nothing.

Member Functions

Command() , FiltersAnyCommand()

```
uint32 Command(void) const
bool FiltersAnyCommand(void) const
```

`Command()` returns the command constant (the BMessage **what** value) that an arriving message must match for the filter to apply. `FiltersAnyCommand()` returns **true** if the filter applies to all messages, and **false** if it's limited to a specific command.

Because all command constants are valid, including negative numbers and 0, `Command()` returns a reliable result only if `FiltersAnyCommand()` returns **false**.

Filter()

```
virtual filter_result Filter(BMessage *message, BHandler **target)
```

Implemented by derived classes to examine an arriving message just before it's dispatched. The first two arguments are the *message* that's being considered, and the proposed BHandler *target*. You can alter the contents of the message, and alter or even replace the handler. If you replace the handler, the new handler must belong to the same looper as the original. The new handler is given an opportunity to filter the message before it's dispatched.

The return value must be one of these two values:

- `B_DISPATCH_MESSAGE`. The message and handler are passed (by the caller) to the looper's `DispatchMessage()` function.
- `B_SKIP_MESSAGE`. The message goes no further—it's immediately thrown away by the caller.

The default version of this function returns `B_DISPATCH_MESSAGE`.

It's possible to call your `Filter()` function yourself (i.e. outside the message-passing mechanism), but keep in mind that it's the caller's responsibility to interpret the return value.

Rather than implement the `Filter()` function, you can supply the BMessageFilter with a `filter_hook` callback when you construct the object. If you do both, the `filter_hook` (and not `Filter()`) will be invoked when the object is asked to examine a message.

FiltersAny see Command()

Looper()

```
```

```
BLooper *Looper(void) const
```

Returns the BLooper whose messages this object filters, or **NULL** if the BMessageFilter hasn't yet been assigned to a BHandler or BLooper. To attach a BMessageFilter to a looper or handler, use **BLooper::AddCommonFilter()** or **BHandler::AddFilter()**.

MessageDelivery() , MessageSource()

```
message_delivery MessageDelivery(void) const
message_source MessageSource(void) const
```

These functions return constants, set when the BMessageFilter object was constructed, that describe the categories of messages that can be filtered. **MessageDelivery()** returns a constant that specifies how the message must be delivered (**B_DROPPED_DELIVERY**, **B_PROGRAMMED_DELIVERY**, or **B_ANY_DELIVERY**). **MessageSource()** returns how the source of the message is constrained (**B_LOCAL_SOURCE**, **B_REMOTE_SOURCE**, or **B_ANY_SOURCE**).

Operators

= (copy)

```
BMessageFilter &operator=(const BMessageFilter&)
```

Copies the filtering criteria and `filter_hook` pointer (if any) from the right-side object into the left-side object.

Constants and Defined Types

filter_hook

```
filter_result (*filter_hook)(BMessage *message,
    BHandler **target,
    BMessageFilter *messageFilter)
```

`filter_hook` defines the protocol for message-filtering functions. The first two arguments are the *message* that's being considered, and the proposed BHandler *target*. You can alter the contents of the message, and alter or even replace the handler. If you replace the handler, the new handler must belong to the same looper as the original. The new handler is given an opportunity to filter the message before it's dispatched.

messageFilter is a pointer to the object on whose behalf this function is being called; you mustn't delete this object. More than one BMessageFilter can use the same `filter_hook` function.

The return value must be one of these two values:

- `B_DISPATCH_MESSAGE`. The message and handler are passed (by the caller) to the looper's `DispatchMessage()` function.
- `B_SKIP_MESSAGE`. The message goes no further—it's immediately thrown away by the caller.

It's possible to call your filter function yourself (i.e. outside the message-passing mechanism), but keep in mind that it's the caller's responsibility to interpret the return value.

You supply a BMessageFilter with a `filter_hook` function when you constuct the object. Alternatively, you can subclass BMessageFilter and provide an implementation of `Filter()`. If you do both, the `filter_hook` (and not `Filter()`) will be invoked when the object is asked to examine a message.

message_source

message_delivery

filter_result

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageFilter Summary

The Application Kit Table of Contents The Application Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BLooper

Derived from: public BHandler
Declared in: be/app/Looper.h
Library: libbe.so
Summary: *more...*

A BLooper object creates a "message loop" that receives messages that are sent or posted to the BLooper. The message loop runs in a separate thread that's spawned (and told to run) when the BLooper receives a `Run()` call. If you're creating your own BLooper, you can invoke `Run()` from within the constructor.

You tell the loop to stop by sending the BLooper a `B_QUIT_REQUESTED` message, which invokes the object's `Quit()` function. You can also call `Quit()` directly, but you have to Lock() the object first (BLooper locking is explained later). `Quit()` deletes the BLooper for you.

The BApplication class, the most important BLooper subclass, bends the above description in a couple of ways:

- A BApplication takes over the main thread, it doesn't spawn a new one.
- You *do* have to **delete** `be_app`; you can't just `Quit()` it.

Messages and Handlers

You can deliver messages to a BLooper's thread by...

- Posting them directly by calling BLooper's `PostMessage()` function.
- Sending them through BMessenger's `SendMessage()` or BMessage's `SendReply()` function.

As messages arrive, they're added to the BLooper's BMessageQueue object. The BLooper takes messages from the queue in the order that they arrived, and calls `DispatchMessage()` for each one. `DispatchMessage()` locks the BLooper and then hands the message to a BHandler object by invoking the handler's `MessageReceived()` function. But which BHandler does the BLooper hand the message to? Here's the path:

- If an incoming message targets a specific BHandler, and if that BHandler is one of the BLooper's *eligible handlers* (as set through the `AddHandler()` function), the BLooper uses that BHandler. (See the BMessage and BMessenger classes for instructions on how to target a BHandler.)
- Otherwise it hands the message to its *preferred handler*, as set through `SetPreferredHandler()`.

- If no preferred handler is set, the BLooper itself handles the message (its own implementation of **`MessageReceived()`** is invoked).

After the handler is finished (when its MessageReceived() returns), the BMessage is automatically deleted and the BLooper is unlocked.

Locking

Access to many BLooper functions (and some BHandler functions) is proteced by a lock. To invoke a lock-protected functions (or groups of functions), you must first call **`Lock()`**, and then call **`Unlock()`** when you're done. The lock is scoped to the calling thread: **`Lock()`**/**`Unlock()`** calls can be nested within the thread. Keep in mind that each **`Lock()`** *must* balanced by an **`Unlock()`**.

The BLooper constructor automatically locks the object. It's unlocked when **`Run()`** is invoked. This means that the **`Run()`** function—and any other lock-protected functions that you call before you call **`Run()`**—must be called from the thread that constructed the BLooper.

Allocation

Because they **`delete`** themselves when told to quit, BLoopers can't be allocated on the stack; you have to construct them with **`new`**.

Hook Functions

- `DispatchMessage()`
- `QuitRequested()`

Constructor and Destructor

BLooper()

```
BLooper(const char *name = NULL,
    int32 priority = B_NORMAL_PRIORITY,
    int32 portCapacity = B_LOOPER_PORT_DEFAULT_CAPACITY)
BLooper(BMessage *archive)
```

Assigns the BLooper object a *name* and then locks it (by calling `Lock()`). *priority* is a value that describes the amount of CPU attention the message loop will receive once it starts running, and *portCapacity* is the number of messages the BLooper can hold in its "message port" (this is *not* the message queue, as explained below).

After you construct the BLooper, you have to tell it to `Run()`. Because the object is locked, `Run()` can only be called from the thread that constructed the object. It's legal to invoke `Run()` from within a subclass implementation of the constructor.

Priority

A set of priority values are defined in **kernel/OS.h**; from lowest to highest, they are:

`B_LOW_PRIORITY`	For threads running in the background that shouldn't interrupt other threads.
`B_NORMAL_PRIORITY`	For all ordinary threads, including the main thread.
`B_DISPLAY_PRIORITY`	For threads associated with objects in the user interface, including window threads.
`B_URGENT_DISPLAY_PRIORITY`	For interface threads that deserve more attention than ordinary windows.
`B_REAL_TIME_DISPLAY_PRIORITY`	For threads that animate the on-screen display.
`B_URGENT_PRIORITY`	For threads performing time-critical computations.
`B_REAL_TIME_PRIORITY`	For threads controlling real-time processes that need unfettered access to the CPUs.

Port Capacity

Messages that are sent to a BLooper first show up in a port (as the term is defined by the Kernel Kit), and then are moved to the BMessageQueue. The capacity of the BMessageQueue is virtually unlimited; the capacity of the port is not. Although messages are moved from the port to the queue as quickly as possible, the port can fill up. A full port will block subsequent message senders.

The default port capacity (100), should be sufficient for most apps, but you can fiddle with it through the *portCapacity* argument.

~BLooper()

virtual **~BLooper()**

Frees the message queue and all pending messages and deletes the message loop. BHandlers that have been added to the BLooper are not deleted, but BMessageFilter objects added as common filters are

In general, you should never **delete** your BLooper objects: With the exception of the BApplication object, BLoopers are destroyed by the **Quit()** function.

If you create a BLooper-derived class that uses multiple inheritance, make sure the first class your mixin class inherits from is BLooper; otherwise, you'll crash when you try to close the window. This happens because of an interaction between the window thread how C++ deletes objects of a multiply-inherited class. In other words:

```
class myClass : public BLooper, public OtherClass {
...
};
```

is safe, while

```
class myClass : public OtherClass, public BLooper {
...
};
```

is not.

Static Functions

LooperForThread()

static BLooper ***LooperForThread**(**thread_id** *thread*)*

Returns the BLooper object that spawned the specified *thread*, or `NULL` if the thread doesn't belong to a BLooper.

Member Functions

AddCommonFilterList() , RemoveCommonFilterList() , SetCommonFilterList() , CommonFilterList()

virtual void **AddCommonFilter(**BMessageFilter ***filter*****)**
virtual bool **RemoveCommonFilter(**BMessageFilter ***filter*****)**
virtual void **SetCommonFilterList(**BList ***filters*****)**
BList ***CommonFilterList(**void**)** const

For all but **CommonFilterList()**, the BLooper must be locked.

These functions manage the BLooper's list of BMessageFilters. Message filters are objects that screen in-coming messages. In the case of BLooper, each message is passed through all filters in the list before it's passed on to **DispatchMessage()**. The order of the filters in the list is determinate. See the BMessageFilter class for details on how message filters work.

AddCommonFilter() adds *filter* to the end of the filter list (creating a BList container if necessary).

RemoveCommonFilter() removes *filter* from the list, but doesn't free the filter. It returns **true** if successful, and **false** if it can't find the specified filter.

SetCommonFilterList() deletes the current filter list and its contents, and replaces it with *filters*. All elements in *filters* must be BMessageFilter pointers. The BLooper takes ownership of all objects in *filters*, as well as *filters* itself. If *filters* is **NULL**, the current list is deleted without a replacement.

CommonFilterList() returns a pointer to the current list. You can examine the list but you shouldn't modify or delete it.

AddHandler() , RemoveHandler() , HandlerAt() , CountHandlers() , IndexOf()

void **AddHandler(**BHandler ***handler*****)**
bool **RemoveHandler(**BHandler ***handler*****)**
BHandler ***HandlerAt(**int32 *index***)** const
int32 **CountHandlers(**void**)** const
int32 **IndexOf(**BHandler ***handler***)** const

AddHandler() adds *handler* to the BLooper's list of BHandler objects, and **RemoveHandler()** removes it. Only BHandlers that have been added to the list are eligible to respond to the messages the BLooper dispatches.

AddHandler() fails if the *handler* already belongs to a BLooper; a BHandler can belong to no more than one BLooper at a time. It can change its affiliation from time to time, but must be removed from one BLooper before it can be added to another. **RemoveHandler()** returns `true` if it succeeds in removing the BHandler from the BLooper, and `false` if not or if the *handler* doesn't belong to the BLooper in the first place.

AddHandler() also calls the *handler*'s **SetNextHandler()** function to assign it the BLooper as its default next handler. **RemoveHandler()** calls the same function to set the *handler*'s next handler to `NULL`.

HandlerAt() returns the BHandler object currently located at *index* in the BLooper's list of eligible handlers, or `NULL` if the index is out of range. Indices begin at 0 and there are no gaps in the list. **CountHandlers()** returns the number of objects currently in the list; the count should always be at least 1, since the list automatically includes the BLooper itself. **IndexOf()** returns the index of the specified *handler*, or **B_ERROR** if that object isn't in the list.

For any of these functions to work, the BLooper must be locked.

See also: **BHandler::Looper()**, **BHandler::SetNextHandler()**, **PostMessage()**, the BMessenger class

---

- Archive() see BArchivable::Archive()

- CommonFilterList() see AddCommonFilterList()
- CountHandlers() see AddHandler()
- CountLockRequests() see LockingThread()
- CountLocks() see LockingThread()

- CurrentMessage() , DetachCurrentMessage()

BMessage ***CurrentMessage**(void) const
BMessage ***DetachCurrentMessage**(void)

The message that a BLooper passes to its handler(s) is called the "current message." These functions access the current message; they're meaningless (they return **NULL**) when called from outside the message processing loop.

`CurrentMessage()` simply returns a pointer to the current message without affecting the BMessage object itself. This is particularly useful to functions that respond to system messages (such as **MouseDown()** and **ScreenChanged()**), but that aren't sent the full BMessage object that initiated the response.

`DetachCurrentMessage()` removes the current message from the message queue and passes ownership of it to the caller; deleting the message is the caller's responsibility. This is useful if you want to delay the response to the message without tying up the BLooper. But be careful—

if the message sender is waiting for a synchronous reply, detaching the message and holding on to it will block the sender.

Detach see CurrentMessage()

DispatchMessage()

```
virtual void DispatchMessage(BMessage *message, BHandler *target)
```

DispatchMessage() is the BLooper's central message-processing function. It's called automatically as messages arrive in the looper's queue, one invocation per message. You never invoke DispatchMessage() yourself.

The default implementation passes *message* to *handler* by invoking the latter's MessageReceived():

```
target->MessageReceived(message);
```

The only exception is where *message*.**what** is B_QUIT_REQUESTED and *handler* is the looper itself; in this case, the object invokes its own QuitRequested() function.

You can override this function to dispatch the messages that your own application defines or recognizes. All unhandled messages should be passed to the base class version, as demonstrated below:

```
void MyLooper::DispatchMessage(BMessage *msg, BHandler *target)
{
   switch ( msg->what ) {
   case MY_MESSAGE1:
      ...
      break;
   case MY_MESSAGE2:
      ...
      break;
   default:
      baseClass::DispatchMessage(msg, target);
      break;
   }
}
```

Also, note that you mustn't delete *message*; it's deleted for you..

The system locks the BLooper before calling DispatchMessage() and keeps it locked for the duration of the function.

HandlerAt() see AddHandler()

- IndexOf() see AddHandler()
- IsLocked() see LockingThread()

Lock() , LockWithTimeout() , Unlock()

bool Lock(void)
status_t LockWithTimeout(bigtime_t *timeout***)**
void Unlock(void)

Lock() locks the BLooper. Locks are held within the context of a thread; while a BLooper is locked, no other thread can invoke its most important functions (**AddHandler()**, **DispatchMessage()**, etc.)

If the looper is already locked (by some other thread), **Lock()** blocks until the looper is unlocked. To set a timeout for the block, use **LockWithTimeout()** instead. *timeout* is measured in microseconds; if it's 0, the function returns immediately (with or without the lock); if it's **B_INFINITE_TIMEOUT**, it blocks without limit.

Unlock() unlocks a locked looper. It can only be called by the thread that currently holds the lock.

Calls to **Lock()**/**LockWithTimeout()** and **Unlock()** can be nested, but locking and unlocking must always be balanced. A single **Unlock()** will *not* undo a series of **Lock()**'s.

BHandler defines "smart" versions of these functions that find the handler's looper and then locks it (or unlocks it) in a pseudo-atomic operation (see **BHandler::LockLooper()**). You should always use the BHandler versions, if possible, rather than retrieving the handler's looper and locking it yourself.

RETURN CODES

Lock() returns **true** if it was able to lock the looper, or if it's already locked by the calling thread, and **false** otherwise.

LockWithTimeout() returns:

- B_OK. The looper was successfully locked.
- **B_TIMED_OUT**. The call timed out without locking the looper.
- **B_BAD_VALUE**. This looper was deleted while the function was blocked.

LockingThread() , IsLocked() , CountLocks() , CountLockRequests() , Sem()

thread_id **LockingThread**(void) const

bool **IsLocked**(void) const
int32 **CountLocks**(void) const
int32 **CountLockRequests**(void) const
sem_id **Sem**(void) const

These functions may be useful while debugging a BLooper.

`LockingThread()` returns the thread that currently has the BLooper locked, or –1 if the BLooper isn't locked.

`IsLocked()` returns `true` if the calling thread currently has the BLooper locked (if it's the locking thread) and `false` if not (if some other thread is the locking thread or the BLooper isn't locked).

`CountLocks()` returns the number of times the locking thread has locked the BLooper—the number of `Lock()` (or `LockWithTimeout()`) calls that have not yet been balanced by matching `Unlock()` calls.

`CountLockRequests()` returns the number of threads currently trying to lock the BLooper. The count includes the thread that currently has the lock plus all threads currently waiting to acquire it.

`Sem()` returns the `sem_id` for the semaphore that the BLooper uses to implement the locking mechanism.

See also: `Lock()`

LockWithTimeout() see Lock()

MessageReceived()

virtual void **MessageReceived**(BMessage **message*)

Simply calls the inherited function. For the current release, the BLooper implementation of this function does nothing of importance.

See also: `BHandler::MessageReceived()`

MessageQueue()

BMessageQueue ***MessageQueue**(void) const

Returns the queue that holds messages delivered to the BLooper's thread. You rarely need to examine the message queue directly; it's made available so you can cheat fate by looking ahead.

See also: the BMessageQueue class

PostMessage()

```
status_t PostMessage(BMessage *message)
status_t PostMessage(uint32 command)
status_t PostMessage(BMessage *message,
    BHandler *handler,
    BHandler *replyHandler = NULL)
status_t PostMessage(uint32 command,
    BHandler *handler,
    BHandler *replyHandler = NULL)
```

PostMessage() is similar to BMessenger::SendMessage(). The BMessenger version is preferred (it's a bit safer than PostMessage()).

Places a message at the far end of the BLooper's message queue. The message will be processed by DispatchMessage() when it comes to the head of the queue.

The message can be a full BMessage object (*message*), or just a command constant (*command*). In the former case, the message is copied and the caller retains ownership of the argument, which can be deleted as soon as PostMessage() returns. In the latter case, a BMessage is created (and deleted) for you.

handler is the designated handler for the message, and must be part of this BLooper's handler chain. If *handler* is (literally) **NULL**, the designated handler is the BLooper's preferred handler at the time DispatchMessage() is called. In the versions of PostMessage() that don't have a *handler* argument, the designated handler is the BLooper object itself.

Replies to the message are delivered to *replyHandler*. If a *replyHandler* isn't specified, replies are sent to be_app_messenger.

A BLooper should never post a message to itself from within its own message loop thread.

RETURN CODES

- B_OK. The message was successfully posted.
- **B_MISMATCHED_VALUES**. *handler* doesn't belong to this BLooper.
- *Other errors*. See the return values for BMessenger::SendMessage().

PreferredHandler() see SetPreferredHandler()

Quit()

virtual void **Quit**(void)

Shuts down the message loop (if it's running), and deletes the BLooper. The object must be locked.

When `Quit()` is called from the BLooper's thread, the message loop is immediately stopped and any messages in the message queue are deleted (without being processed). Note that, in this case, `Quit()` doesn't return since the calling thread is dead.

When called from another thread, `Quit()` waits until all messages currently in the queue have been handled before it kills the message loop. It returns after the BLooper has been deleted.

If you're quitting a BLooper from some other thread, you should send the object a `B_QUIT_REQUESTED` message rather than calling `Quit()` directly.

QuitRequested()

virtual **bool QuitRequested(void)**

Hook function that's invoked when the BLooper receives a `B_QUIT_REQUESTED` message. You never invoke this function directly. Derived classes implement this function to return **true** if it's okay to quit this BLooper, and **false** if not. Note that this function does *not* actually quit the object—the code that handles the `B_QUIT_REQUESTED` message does that.

BLooper's default implementation of `QuitRequested()` always returns **true**.

RemoveCommonFilter() see AddCommonFilterList()

Run()

virtual **thread_id Run(void)**

Spawns the message loop thread and starts it running. `Run()` expects the BLooper to be locked (once only!) when it's called; it unlocks the object before it returns. Keep in mind that a BLooper is locked when it's constructed.

Calling `Run()` on a BLooper that's already running will dump you into the debugger.

RETURN CODES

- *Positive values*. The thread was successfully spawned and started; this is the thread_id for the thread.
- *Thread errors*. See `spawn_thread()` and `resume_thread()`.

- *Port errors*. See **create_port()**.

AddCommonFilter()

SetPreferredHandler() , PreferredHandler()

void **SetPreferredHandler(**BHandlerAddCommonFilter() **handler**) const
BHandler ***PreferredHandler(**void**)**

These functions set and return the BLooper's preferred handler—the BHandler object that should handle messages not specifically targetted to another BHandler.

To designate the current preferred handler—whatever object that may be—as the target of a message, pass NULL for the target handler to **PostMessage()** or to the BMessenger constructor.

Posting or sending messages to the preferred handler can be useful. For example, in the Interface Kit, BWindow objects name the current focus view as the preferred handler. This makes it possible for other objects—such as BMenuItems and BButtons—to target messages to the BView that's currently in focus, without knowing what view that might be. For example, by posting its messages to the window's preferred handler, a Cut menu item can make sure that it always acts on whatever view contains the current selection. See the chapter on the Interface Kit for information on windows, views, and the role of the focus view.

By default, BLoopers don't have a preferred handler; until one is set, **PreferredHandler()** returns **NULL**. Note however, that messages targeted to the preferred handler are dispatched to the BLooper whenever the preferred handler is NULL. In other words, the BLooper acts as default preferred handler, even though the default is formally NULL.

See also: **BControl::SetTarget()** and **BMenuItem::SetTarget()** in the Interface Kit, **PostMessage()**

Thread() , Team()

thread_id **Thread(**void**)** const
team_id **Team(**void**)** const

These functions identify the thread that runs the message loop and the team to which it belongs. **Thread()** returns **B_ERROR** if **Run()** hasn't yet been called to spawn the thread and begin the loop. **Team()** always returns the application's **team_id**.

Unlock() see Lock()

Constants

B_LOOPER_PORT_DEFAULT_CAPACITY

```
#define B_LOOPER_PORT_DEFAULT_CAPACITY 100
```

The default capacity of the port that holds incoming messages before they're placed in the BLooper's BMessageQueue. The capacity is set in the BLooper constructor.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BHandler

Derived from: none
Mix-in classes: BArchivable
Declared in: be/app/Handler.h
Library: libbe.so
Summary: *more...*

A BHandler object responds to messages that are handed to it by a BLooper. The BLooper tells the BHandler about a message by invoking the BHandler's MessageReceived() function.

The Handler List

To be eligible to get messages from a BLooper, a BHandler must be in the BLooper's list of *eligible handlers* (as explained in the BLooper class). The list of eligible handlers is ordered; if the "first" handler doesn't want to respond to a message that it has received, it simply calls the inherited version of **`MessageReceived()`** and the message will automatically be handed to the object's "next" handler. (System messages are not handed down the list.) The BLooper that all these BHandlers belong to is always the last the last handler in the list (BLooper inherits from BHandler).

A BHandler's next handler assignment can be changed through **`SetNextHandler()`**.

Targets

You can designate a target BHandler for most messages. The designation is made when calling BLooper's **`PostMessage()`** function or when constructing the BMessenger object that will send the message. Messages that a user drags and drops are targeted to the object (a BView) that controls the part of the window where the message was dropped. The messaging mechanism eventually passes the target BHandler to **`DispatchMessage()`**, so that the message can be delivered to its designated destination.

Filtering

Messages can be filtered before they're dispatched—that is, you can define a function that will look at the message before the target BHandler's hook function is called. The filter function is associated with a BMessageFilter object, which records the criteria for calling the function.

Filters that should apply only to messages targeted to a particular BHandler are assigned to the BHandler by **`SetFilterList()`** or **`AddFilter()`**. Filters that might apply to any message a BLooper dispatches, regardless of its target, are assigned by the parallel BLooper functions, **`SetCommonFilterList()`** and **`AddCommonFilter()`**. See those functions and the BMessageFilter class for details.

Notifiers and Observers

A BHandler can be a **notifier**. A notifier is a handler that maintains one or more states and notifies interested parties when those states change. Each state is idenfified by a 32-bit "what" code. Interested parties, called **observers**, can register to monitor changes in one or more states by calling `StartWatching()` and specifying the "what" code of the state they want to be notified of changes to.

This notification occurs when the BHandler calls `SendNotices()`; it's the handler's job to call `SendNotices()` whenever a state changes, to ensure that observers are kept informed of the changes. The BHandler passes to `SendNotices()` a message template to be sent to the observers.

When a notification is sent, observers receive a `B_OBSERVER_NOTICE_CHANGE` message with an int32 field **`B_OBSERVE_WHICH_CHANGE`** that contains the "what" code of the state that changed, and a **`B_OBSERVE_ORIGINAL_WHAT`** field that contains the "what" value that was in the template BMessage.

Hook Functions

MessageReceived()

Constructor and Destructor

BHandler()

```
BHandler(const char *name = NULL)
BHandler(BMessage *archive)
```

Initializes the BHandler by assigning it a *name* and registering it with the messaging system. BHandlers can also be reconstructed from a BMessage *archive*.

~BHandler()

```
virtual ~BHandler()
```

Deletes any BMessageFilters assigned to the BHandler.

Static Functions

- Instantiate() see BArchivable::Instantiate()

Member Functions

- AddFilter() see SetFilterList()

- Archive() , see

- FilterList() see SetFilterList()

- GetSupportedSuites()

```
virtual status_t GetSupportedSuites(BMessage *message)
```

Implemented by derived classes to report the suites of messages and specifiers they understand. This function is called in response to either a **B_GET_PROPERTIES** scripting message for the "Suites" property or a **B_GET_SUPPORTED_SUITES** message.

Each derived class should add the names of the suites it implements to the "suites" array of *message*. Each item in the array is a MIME string with the "suite" supertype. In addition, the class should add corresponding flattened BPropertyInfo objects in the "messages" array. A typical implementation of **GetSupportedSuites()** looks like:

```
status_t MyHandler::GetSupportedSuites(BMessage *message)
{
   message->AddString("suites", "suite/vnd.Me-my_handler"));
   BPropertyInfo prop_info(prop_list);
   message->AddFlat("messages", &prop_info);
   return BHandler::GetSupportedSuites(message);
}
```

The value returned by **GetSupportedSuites()** is added to *message* in the int32 "error" field.

BHandler's version of this function adds the universal suite "suite/vnd.Be-handler" to *message* then returns **B_OK**.

- LockLooper() , LockLooperWithTimeout() , UnlockLooper()

```
bool LockLooper(void)
status_t LockLooperWithTimeout(bigtime_t timeout)
void UnlockLooper(void)
```

These are "smart" versions of BLooper's locking functions (**BLooper::Lock()** et. al.). The difference between the versions is that these functions retrieve the handler's looper and lock

it (or unlock it) in a pseudo-atomic operation, thus avoiding a race condition. Anytime you're tempted to write code such as this:

```
/* DON'T DO THIS */
if (myHandler->Looper()->Lock()) {
   ...
   myHandler->Looper()->Unlock();
}
```

Don't do it. Instead, do this:

```
/* DO THIS INSTEAD */
if (myHandler->LockLooper()) {
   ...
   myHandler->UnlockLooper();
}
```

Except for an additional return value in **LockLooperWithTimeout()**, these functions are identical to their BLooper analogues. See to **BLooper::Lock()** for details.

RETURN CODES

LockLooper() returns **true** if it was able to lock the looper, or if it's already locked by the calling thread, and **false** otherwise. If the handler changes loopers during the call, **false** is returned.

LockLooperWithTimeout() returns:

- B_OK. The looper was successfully locked.
- **B_TIMED_OUT**. The call timed out without locking the looper.
- **B_BAD_VALUE**. This handler's looper is invalid.
- **B_MISMATCHED_VALUES**. The handler switched loopers during the call.

LockLooperWithTimeout() see LockLooper()

Looper()

```
BLooper *Looper(void) const
```

Returns the BLooper object that the BHandler has been added to. The function returns **NULL** if the object hasn't been added to a BLooper. A BHandler can be associated with only one BLooper at a time.

Note that a BLooper object automatically adds itself (as a handler) to itself (as a looper), and a BWindow automatically adds its child views. To explicitly add a handler to a looper, you call **BLooper::AddHandler()**.

MessageReceived()

virtual void MessageReceived(BMessage **message***)**

Implemented by derived classes to respond to messages that are received by the BHandler. The default (BHandler) implementation of this function responds only to scripting requests. It passes all other messages to the next handler by calling that object's version of **MessageReceived()**.

A typical **MessageReceived()** implementation distinguishes between messages by looking at its command constant (i.e. the **what** field). For example:

```
void MyHandler::MessageReceived(BMessage *message)
{
   switch ( message->what ) {
   case COMMAND_ONE:
      HandleCommandOne()
      break;
   case COMMAND_TWO:
      HandleCommandTwo()
      break;
   ...
   default:
      baseClass::MessageReceived(message);
      break;
   ...
   }
}
```

It's essential that all unhandled messages are passed to the base class implementation of **MessageReceived()**, as shown here. The handler chain model depends on it.

If the message comes to the end of the line—if it's not recognized and there is no next handler—the BHandler version of this function sends a **B_MESSAGE_NOT_UNDERSTOOD** reply to notify the message source.

*Do not delete th**e argument** message when you're done with.* It doesn't belong to you.

Name() see SetName()

NextHandler() see SetNextHandler()

ResolveSpecifier()

```
virtual BHandler *ResolveSpecifier(BMessage *message,
    int32 index,
    BMessage *specifier,
    int32 what,
    const char *property)
```

Implemented by derived classes to determine the proper handler for a scripting message. The message is targeted to the BHandler, but the specifiers may indicate that it should be assigned to another object. It's the job of **ResolveSpecifier()** to examine the current specifier (or more, if necessary) and return the object that should either handle the message or look at the next specifier. This function is called before the message is dispatched and before any filtering functions are called.

The first argument, *message*, points to the scripting message under consideration. The current specifier is passed in *specifier*; it will be at index *index* in the specifier array of *message*. Finally, *what* contains the **what** data member of *specifier* while *property* contains the name of the targetted property.

ResolveSpecifier() returns a pointer to the next BHandler that should look at the message. To identify the BHandler, it tries these methods, in order:

Method 1: If the *specifier* identifies a BHandler belonging to another BLooper, it should send the *message* to the BLooper and return `NULL`. The message will be handled in the message loop of the other BLooper; it won't be further processed in this one. For example, a BHandler that kept a list of proxies might use code like the following:

```
if ( (strcmp(property, "Proxy") == 0)
            && (what == B_INDEX_SPECIFIER) ) {
   int32 i;
   if ( specifier->FindInt32("index", &i) == B_OK ) {
      MyProxy *proxy = (MyProxy *)proxyList->ItemAt(i);
      if ( proxy ) {
            message->PopSpecifier();
            if ( proxy->Looper() != Looper() ) {
               proxy->Looper()->PostMessage(message, proxy);
               return NULL;
            }
      }
      . . .
   }
   . . .
}
```

Since this function resolved the specifier at *index*, it calls **PopSpecifier()** to decrement the index before forwarding the message. Otherwise, the next handler would try to resolve the same specifier.

Method 2: If the *specifier* picks out another BHandler object belonging to the same BLooper, **ResolveSpecifier()** can return that BHandler. For example:

```
if ( proxy ) {
   message->PopSpecifier();
   if ( proxy->Looper() != Looper() ) {
      proxy->Looper()->PostMessage(message, proxy);
      return NULL;
   }
   else {
      return proxy;
   }
}
```

This, in effect, puts the returned object in the BHandler's place as the designated handler for the message. The BLooper will give the returned handler a chance to respond to the message or resolve the next specifier.

Again, **PopSpecifier()** should be called so that an attempt isn't made to resolve the same specifier twice.

Method 3: If it can resolve all remaining specifiers and recognizes the message as one that the BHandler itself can handle, it should return the BHandler (`this`). For example:

```
if ( (strcmp(property, "Value") == 0) && (message->what == B_GET_PROPERTY) )
   return this;
```

This confirms the BHandler as the message target. **ResolveSpecifier()** won't be called again, so it's not necessary to call **PopSpecifier()** before returning.

Method 4: If it doesn't recognize the property or can't resolve the specifier, it should call (and return the value returned by) the inherited version of **ResolveSpecifier()**.

Examples

The BApplication object takes the first path when it resolves a specifier for a "Window" property; it sends the message to the specified BWindow and returns `NULL`. A BWindow follows the second path when it resolves a specifier for a "View" property; it returns the specified BView. Thus, a message initially targeted to the BApplication object can find its way to a BView.

BHandler's version of **ResolveSpecifier()** recognizes a **B_GET_PROPERTY** *message* with a direct *specifier* requesting a "Suite" for the supported suites, "Messenger" for the BHandler, or the BHandler's "InternalName" (the same name that its **Name()** function returns). In all three cases, it assigns the BHandler (`this`) as the object responsible for the message.

For all other specifiers and messages, it sends a **B_MESSAGE_NOT_UNDERSTOOD** reply and returns `NULL`. The reply message has an "error" field with **B_SCRIPT_SYNTAX** as the error and a "message" field with a longer textual explanation of the error.

SetFilterList() , FilterList() , AddFilter() , RemoveFilter()

virtual void **SetFilterList(**BList **list***)**
BList ***FilterList**(void) const
virtual void **AddFilter(**BMessageFilter **filter***)**
virtual bool **RemoveFilter(**BMessageFilter **filter***)**

These functions manage a list of BMessageFilter objects associated with the BHandler.

SetFilterList() assigns the BHandler a new *list* of filters; the list must contain pointers to instances of the BMessageFilter class or to instances of classes that derive from BMessageFilter. The new list replaces any list of filters previously assigned. All objects in the previous list are deleted, as is the BList that contains them. If *list* is **NULL**, the current list is removed without a replacement. **FilterList()** returns the current list of filters.

AddFilter() adds a *filter* to the end of the BHandler's list of filters. It creates the BList object if it doesn't already exist. By default, BHandlers don't maintain a BList of filters until one is assigned or the first BMessageFilter is added. **RemoveFilter()** removes a *filter* from the list without deleting it. It returns `true` if successful, and `false` if it can't find the specified filter in the list (or the list doesn't exist). It leaves the BList in place even after removing the last filter.

For **SetFilterList()**, **AddFilter()**, and **RemoveFilter()** to work, the BHandler must be assigned to a BLooper object and the BLooper must be locked.

See also: **BLooper::SetCommonFilterList()**, **BLooper::Lock()**, the BMessageFilter class

SetName() , Name()

void **SetName(**const char **string***)**
const char ***Name**(void) const

These functions set and return the name that identifies the BHandler. The name is originally set by the constructor. **SetName()** assigns the BHandler a new name, and **Name()** returns the current name. The string returned by **Name()** belongs to the BHandler object; it shouldn't be altered or freed.

See also: the BHandler constructor, **BView::FindView()** in the Interface Kit

SetNextHandler() , NextHandler()

void **SetNextHandler(**BHandler **handler***)**
BHandler ***NextHandler**(void) const

SetNextHandler() reorders the objects in the *handler* chain so that handler follows this BHandler. This BHandler and *handler* must already be part of the same chain, and the BLooper they belong to must be locked. The order of objects in the handler chain affects the way incoming messages are handled (as explained in "Inheritance and the Handler Chain". By default handlers are placed in the order that they're added (through **BLooper::AddHandler()**).

NextHandler() returns this object next handler. If this object is at the end of the chain, it returns **NULL**.

StartWatching() , StartWatchingAll() , StopWatching() , StopWatchingAll()

status_t **StartWatching(**BMessenger *watcher*, uint32 *what***)** status_t **StartWatching(**BHandler **watcher*, uint32 *what***)**
status_t **StartWatchingAll(**BMessenger *watcher***)** status_t **StartWatchingAll(**BHandller **watcher***)**
status_t **StopWatching(**BMessenger *watcher*, uint32 *what***)** status_t **StopWatching(**BHandler **watcher*, uint32 *what***)**
status_t **StopWatchingAll(**BMessenger *watcher***)** status_t **StopWatchingAll(**BHandller **watcher***)**

The BHandler class provides the concept of a **notifier**. Notifiers maintain one or more states that other entities might want to monitor changes to. These states are identified by a 32-bit *what* code. Another entity—a BHandler or a BMessenger—can watch for changes notifiers' states. These are called **observers**.

StartWatching() registers the BMessenger or BHandler specified by *watcher* to be notified whenever the state specified by *what* changes. **StartWatchingAll()** registers the specified BMessenger or BHandler to be notified when any of the notifer's states change.

StartWatching() works by sending a message to the BHandler you want to observe, with a BMessenger back to the observer, so both must be attached to a looper at the time **StartWatching()** is called.

The forms of **StartWatching()** and **StartWatchingAll()** that accept a BHandler can be used to observe a handler that's not yet attached to a looper. However, these only work if the observer and notifier are both in the same looper.

StopWatching() ceases monitoring of the state *what*. **StopWatchingAll()**, by some odd coincidence, stops all monitoring by the BHandler or BMessenger specified by *watcher*.

RETURN CODES

- **B_OK**. No error.
- **B_BAD_HANDLER**. The specified BHandler isn't valid.

- UnlockLooper() see LockLooper()

Archived Fields

Field	Type code	Meaning
"_name"	B_STRING_TYPE	The object's name (see SetName()).

BHandler records its own name.

Scripting Suites and Properties

Suite: suite/vnd.Be-handler

"InternalName"

Message	Specifiers	Reply Type
B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_STRING_TYPE

Returns the handler's name.

"Messenger"

Message	Specifiers	Reply Type
B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_MESSENGER_TYPE

Returns a BMessenger for the handler.

"Suites"

Message	Specifiers	Reply Type
B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_STRING_TYPE array

Returns an array of suites that the target supports, identified by name (e.g. "suite/vnd.Be-handler").

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BArchivable

Derived from: *none*
Declared in: be/support/Archivable.h
Summary: *more...*

BArchivable is a protocol for *archiving* and *unarchiving* objects. When you archive an object, you record its state into a BMessage that can be sent to another application, flattened and saved as a file, cached in memory, and so on. Unarchiving does the opposite: It takes a BMessage archive and turns it into a functioning object.

See "Archiving" for more information on the BeOS archival system.

Constructor and Destructor

BArchivable()

```
BArchivable(void)
BArchivable(BMessage *archive)
```

Does nothing.

~BArchivable()

```
virtual ~BArchivable()
```

Does nothing.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*****)**

Derived classes should implement `Instantiate()` to return a new BArchivable object that was constructed from the BMessage *archive*. For example:

```
BArchivable *TheClass::Instantiate(BMessage *archive)
{
   if ( !validate_instantiation(archive, "TheClass") )
      return NULL;
   return new TheClass(archive);
}
```

`Instantiate()` must return a BArchivable *, regardless of the actual class in which it's implemented.

This function depends on a constructor that can initialize the new object from the *archive* BMessage. See "Instantiability" for more information.

The default implementation returns `NULL`.

Member Functions

Archive()

virtual status_t **Archive(**BMessage ***archive**, bool *deep* = true**)** const

The default implementation adds the name of the object's class to *archive*'s "class" field. Derived classes must override **Archive()** to augment this implementation by adding, to the BMessage, data that describes the current state of the object. Each implementation of this function should begin by incorporating the inherited version:

```
/* We'll assume that MyView inherits from BView. */
status_t MyView::Archive(BMessage *archive, bool deep)
{
   BView::Archive(archive, deep);
   . . .
}
```

If the class can be instantiated directly from a derived class, it should also add its name to the "class" array:

```
archive->AddString("class", "MyView");
```

The *deep* flag declares whether **Archive()** should include objects that "belong" to the archiving object. For example, a deep BView archive would include archived forms of the view's children.

Archive() should return **B_OK** if it's successful; otherwise, it should return **B_ERROR** or a more descriptive error code.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Support Kit Table of Contents

- The Support Kit
- Archiving
- BArchivable
- BAutolock
- BBlockCache
- BDataIO
- BFlattenable
- BList
- BLocker
- BMallocIO
- BMemoryIO
- BMessageRunner
- BPositionIO
- BStopWatch
- BString
- Thread Local Storage
- Functions and Macros
- Debugging Tools
- Constants and Defined Types
- Error Codes

The Be Book **The Support Kit Index**

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Support Kit: Master Index

!

!=	BString

&

>	BString
>=	BString
<	BString
<<	BString
<=	BString

+

+=	BString

=

=	BList
=	BString
==	BString

[

A

AddItem()	BList
AddList()	BList
add_system_beep_event()	Functions and Macros
Adopt()	BString
AllowsTypeCode()	BFlattenable
B_ALREADY_RUNNING	Error Codes
B_AMBIGUOUS_APP_LAUNCH	Error Codes
An illegal argument type was named or passed	Error Codes
An object or structure isn't properly initialized	Error Codes
An unsupported operation was	Error Codes
B_ANY_TYPE	Constants and Defined Types
Append()	BString
Application Kit Error Codes	Error Codes
BArchivable	BArchivable
BArchivable()	BArchivable
~BArchivable()	BArchivable
Archive()	BArchivable
Archiving and Unarchiving	Archiving
Archiving	Archiving

Archiving	Archiving
ASSERT_WITH_MESSAGE()	Debugging Tools
Assigning and Retrieving String Data	BString
atomic_add()	Functions and Macros
atomic_and()	Functions and Macros
atomic_or()	Functions and Macros
BAutolock	BAutolock
BAutolock()	BAutolock
~BAutolock()	BAutolock

B

B_BUSY	Error Codes
B_BAD_ADDRESS	Error Codes
B_BAD_HANDLER	Error Codes
B_BAD_IMAGE_ID	Error Codes
B_BAD_INDEX	Error Codes
B_BAD_MIME_SNIFFER_RULE	Error Codes
B_BAD_PORT_ID	Error Codes
B_BAD_REPLY	Error Codes
B_BAD_SCRIPT_SYNTAX	Error Codes
B_BAD_SEM_ID	Error Codes
B_BAD_TEAM_ID	Error Codes
B_BAD_THREAD_ID	Error Codes
B_BAD_THREAD_STATE	Error Codes
B_BAD_TYPE	Error Codes

B_BAD_VALUE	Error Codes
beep()	Functions and Macros
bigtime_t	Constants and Defined Types
BBlockCache	BBlockCache
BBlockCache()	BBlockCache
~BBlockCache()	BBlockCache
bool	Constants and Defined Types
B_BOOL_TYPE	Constants and Defined Types
Boolean Constants	Constants and Defined Types
Buffer()	BMallocIO
BufferLength()	BMallocIO
B_BUSTED_PIPE	Error Codes
ByteAt()	BString
Byte Swapping Constants	Constants and Defined Types

C

Capitalize()	BString
CapitalizeEachWord()	BString
cast_as()	Functions and Macros
B_CHAR_TYPE	Constants and Defined Types
CharacterDeescape()	BString
CharacterEscape()	BString
class_name()	Functions and Macros
B_COLOR_8_BIT_TYPE	Constants and Defined

	Types
Compare()	BString
Conflicting values were passed to the function	Error Codes
Constants and Defined Types	Constants and Defined Types
Constants and Defined Types	Constants and Defined Types
Constants	Constants and Defined Types
Constructor and Destructor	BArchivable
Constructor and Destructor	BAutolock
Constructor and Destructor	BBlockCache
Constructor and Destructor	BDataIO
Constructor and Destructor	BList
Constructor and Destructor	BLocker
Constructor and Destructor	BMallocIO
Constructor and Destructor	BMemoryIO
Constructor and Destructor	BMessageRunner
Constructor and Destructor	BPositionIO
Constructor and Destructor	BStopWatch
Constructor and Destructor	BString
A convenient catchall for general errors	Error Codes
convert_from_utf8()	Functions and Macros
convert_to_utf8()	Functions and Macros
CopyInto()	BString

CountChars()	BString
CountItems()	BList
CountLockRequests()	BLocker
CountLocks()	BLocker
B_CROSS_DEVICE_LINK	Error Codes

D

DEBUGGER()	Debugging Tools
The DEBUG Preprocessor Variable	Debugging Tools
BDataIO	BDataIO
BDataIO()	BDataIO
~BDataIO()	BDataIO
The Debug Flag	Debugging Tools
DEBUG_ONLY()	Debugging Tools
B_DEBUGGER_ALREAD	Error Codes
Debugging Tools	Debugging Tools
Debugging Tools	Debugging Tools
Defined Types	Constants and Defined Types
B_DEV_BAD_DRIVE_NUM	Error Codes
B_DEV_CONFIGURATION_ERROR	Error Codes
B_DEV_DISABLED_BY_USER	Error Codes
B_DEV_DOOR_OPEN	Error Codes
B_DEV_FORMAT_ERROR	Error Codes
B_DEV_ID_ERROR	Error Codes

B_DEV_INVALID_IOCTL	Error Codes
B_DEV_MEDIA_CHANGE_REQUESTED	Error Codes
B_DEV_NO_MEDIA	Error Codes
B_DEV_NO_MEMORY	Error Codes
B_DEV_NOT_READY	Error Codes
B_DEV_READ_ERROR	Error Codes
B_DEV_RECALIBRATE_ERROR	Error Codes
B_DEV_RESOURCE_CONFLICT	Error Codes
B_DEV_SEEK_ERROR	Error Codes
B_DEV_TIMEOUT	Error Codes
B_DEV_UNREADABLE	Error Codes
B_DEV_WRITE_ERROR	Error Codes
Device Kit Errors	Error Codes
B_DEVICE_FULL	Error Codes
Direct Data Access	BString
B_DIRECTORY_NOT_EMPT	Error Codes
DoForEach()	BList
B_DOUBLE_TYPE	Constants and Defined Types
B_DUPLICATE_REPLY	Error Codes
Dynamic Loading	Archiving

E

EACCESS	Error Codes
EAGAIN	Error Codes

EBADF	Error Codes
EBUSY	Error Codes
EEXIST	Error Codes
EINTR	Error Codes
EINVAL	Error Codes
EIO	Error Codes
EISDIR	Error Codes
ELOOP	Error Codes
EMFILE	Error Codes
ENAME	Error Codes
ENOENT	Error Codes
ENOMEM	Error Codes
ENOSPC	Error Codes
ENOTDIR	Error Codes
ENOTEMPTY	Error Codes
EPERM	Error Codes
EPIPE	Error Codes
EROFS	Error Codes
B_ERROR	Error Codes
ETIMEDOUT	Error Codes
EWOULD	Error Codes
EXDEV	Error Codes
ElapsedTime()	BStopWatch
Empty String	Constants and Defined Types
Encoding Conversion Constants	Constants and Defined Types
B_ENTRY_NOT_FOUND	Error Codes

Error Codes	Error Codes
Error Codes	Error Codes
B_EUC_CONVERSION	Constants and Defined Types
Everything's OK	Error Codes

F

false	Constants and Defined Types
File System Error Codes	Error Codes
B_FILE_ERROR	Error Codes
B_FILE_EXISTS	Error Codes
FindFirst()	BString
FindLast()	BString
find_instantiation_func()	Functions and Macros
FirstItem()	BList
Flatten()	BFlattenable
BFlattenable	BFlattenable
BFlattenable	BFlattenable
FlattenedSize()	BFlattenable
B_FLOAT_TYPE	Constants and Defined Types
Function Pointers	Constants and Defined Types
Functions and Macros	Functions and Macros
Functions and Macros	Functions and Macros

G

General Error Codes	Error Codes
Get()	BBlockCache
GetInfo()	BMessageRunner
B_GRAYSCALE_8_BIT_TYPE	Constants and Defined Types

H

HasItem()	BList

I

ICompare()	BString
IFindFirst()	BString
IFindLast()	BString
B_INTERRUPTED	Error Codes
IReplace()	BString
IReplaceFirst()	BString
IReplaceLast()	BString
The index is out of range	Error Codes
IndexOf()	BList
InitCheck()	BMessageRunner
Insert()	BString
Instantiability	Archiving
Instantiate()	BArchivable
instantiate_object()	Functions and Macros
instantiation_func	Constants and Defined Types

B_INT16_TYPE	Constants and Defined Types
B_INT32_TYPE	Constants and Defined Types
B_INT64_TYPE	Constants and Defined Types
B_INT8_TYPE	Constants and Defined Types
Integer Types	Constants and Defined Types
B_IO_ERROR	Error Codes
IsEmpty()	BList
IsFixedSize()	BFlattenable
IsLocked()	BAutolock
IsLocked()	BLocker
B_IS_A_DIRECTORY	Error Codes
IS_DEBUG_ENABLED()	Debugging Tools
is_instance_of()	Functions and Macros
is_kind_of()	Functions and Macros
is_type_swapped()	Functions and Macros
B_ISO10_CONVERSION	Constants and Defined Types
B_ISO1_CONVERSION	Constants and Defined Types
B_ISO2_CONVERSION	Constants and Defined Types
B_ISO3_CONVERSION	Constants and Defined Types
B_ISO4_CONVERSION	Constants and Defined Types
B_ISO5_CONVERSION	Constants and Defined Types
B_ISO6_CONVERSION	Constants and Defined Types
B_ISO7_CONVERSION	Constants and Defined Types
B_ISO8_CONVERSION	Constants and Defined Types
B_ISO9_CONVERSION	Constants and Defined Types
ItemAt()	BList

Items()	BList

K

Kernel Kit Error Codes	Error Codes

L

Lap()	BStopWatch
LastItem()	BList
B_LAUNCH_FAILED	Error Codes
B_LAUNCH_FAILED_APP_IN_TRASH	Error Codes
B_LAUNCH_FAILED_APP_NOT_FOUND	Error Codes
B_LAUNCH_FAILED_EXECUTABLE	Error Codes
B_LAUNCH_FAILED_FILES_APP_NOT_F	Error Codes
B_LAUNCH_FAILED_NO_PREFERRED_A	Error Codes
B_LAUNCH_FAILED_NO_RESOLVE_LINK	Error Codes
Length()	BString
B_LINK_LIMIT	Error Codes
BList	BList
BList()	BList
~BList()	BList
Lock()	BLocker
LockBuffer()	BString
LockWithTimeout()	BLocker
BLocker	BLocker

BLocker()	BLocker
~BLocker()	BLocker
LockingThread()	BLocker

M

B_MAC_ROMAN_CONVERSION	Constants and Defined Types
Macros	Debugging Tools
Mail Errors	Error Codes
B_MAIL_ACCESS_ERROR	Error Codes
B_MAIL_INVALID_MAIL	Error Codes
B_MAIL_NO_DAEMON	Error Codes
B_MAIL_NO_RECIPIENT	Error Codes
B_MAIL_UNKNOWN_FIELD	Error Codes
B_MAIL_UNKNOWN_HOST	Error Codes
B_MAIL_UNKNOWN_USER	Error Codes
B_MAIL_WRONG_PASSWOR	Error Codes
MakeEmpty()	BList
BMallocIO	BMallocIO
BMallocIO()	BMallocIO
~BMallocIO()	BMallocIO
max()	Functions and Macros
max_c()	Functions and Macros
Media Kit Error Codes	Error Codes
B_MEDIA_ADDON_DISABLED	Error Codes
B_MEDIA_ADDON_FAILED	Error Codes

B_MEDIA_ADDON_RESTRICT	Error Codes
B_MEDIA_ALREADY_CONNE	Error Codes
B_MEDIA_APP_ALREADY_RE	Error Codes
B_MEDIA_APP_NOT_REGIST	Error Codes
B_MEDIA_BAD_CLIP_FORMA	Error Codes
B_MEDIA_BAD_DESTINATION	Error Codes
B_MEDIA_BAD_FORMAT	Error Codes
B_MEDIA_BAD_NODE	Error Codes
B_MEDIA_BAD_SOURCE	Error Codes
B_MEDIA_BUFFER_ALREADY	Error Codes
B_MEDIA_BUFFERS_NOT_RE	Error Codes
B_MEDIA_CANNOT_CHANGE	Error Codes
B_MEDIA_CANNOT_RECLAIM	Error Codes
B_MEDIA_CANNOT_SEEK	Error Codes
B_MEDIA_CHANGE_IN_PROG	Error Codes
B_MEDIA_DUPLICATE_FORM	Error Codes
B_MEDIA_NO_HANDLER	Error Codes
B_MEDIA_NODE_ALREADY_E	Error Codes
B_MEDIA_NODE_BUSY	Error Codes
B_MEDIA_NOT_CONNECTED	Error Codes
B_MEDIA_PARAMETER_GROUP	Constants and Defined Types
B_MEDIA_PARAMETER_TYPE	Constants and Defined Types
B_MEDIA_PARAMETER_WEB_T	Constants and Defined Types
B_MEDIA_REALTIME_DISABL	Error Codes
B_MEDIA_REALTIME_UNAVAI	Error Codes
B_MEDIA_STALE_CHANGE_C	Error Codes

B_MEDIA_SYSTEM_FAILURE	Error Codes
B_MEDIA_TIME_SOURCE_BU	Error Codes
B_MEDIA_TIME_SOURCE_ST	Error Codes
B_MEDIA_TOO_MANY_BUFF	Error Codes
B_MEDIA_TOO_MANY_NODE	Error Codes
Member Functions	BArchivable
Member Functions	BAutolock
Member Functions	BBlockCache
Member Functions	BDataIO
Member Functions	BFlattenable
Member Functions	BList
Member Functions	BLocker
Member Functions	BMallocIO
Member Functions	BMemoryIO
Member Functions	BMessageRunner
Member Functions	BPositionIO
Member Functions	BStopWatch
Member Functions	BString
BMemoryIO	BMemoryIO
BMemoryIO()	BMemoryIO
~BMemoryIO()	BMemoryIO
BMessageRunner	BMessageRunner
BMessageRunner()	BMessageRunner
~BMessageRunner()	BMessageRunner
B_MESSAGE_TO_SELF	Error Codes
B_MESSAGE_TYPE	Constants and Defined Types

B_MESSENGER_TYPE	Constants and Defined Types
B_MIME_TYPE	Constants and Defined Types
min()	Functions and Macros
min_c()	Functions and Macros
B_MISMATCHED_VALUE	Error Codes
B_MISSING_LIBRARY	Error Codes
B_MISSING_SYMBOL	Error Codes
B_MONOCHROME_1_BIT_TYPE	Constants and Defined Types
MoveInto()	BString

N

NIL	Constants and Defined Types
NULL and NIL	Constants and Defined Types
NULL	Constants and Defined Types
Name()	BStopWatch
B_NAME_IN_USE	Error Codes
B_NAME_NOT_FOUND	Error Codes
B_NAME_TOO_LONG	Error Codes
B_NO_ERROR	Error Codes
B_NO_INIT	Error Codes
B_NO_MEMORY	Error Codes
B_NO_MORE_FDS	Error Codes
B_NO_MORE_PORTS	Error Codes
B_NO_MORE_SEMS	Error Codes
B_NO_MORE_TEAMS	Error Codes

B_NO_MORE_THREADS	Error Codes
B_NOT_A_DIRECTORY	Error Codes
B_NOT_ALLOWED	Error Codes
B_NOT_AN_EXECUTABL	Error Codes

O

B_OK	Error Codes
B_OBJECT_TYPE	Constants and Defined Types
B_OFF_T_TYPE	Constants and Defined Types
Operators	BList
Operators	BString

P

POSIX	Error Codes
POSIX Errors	Error Codes
PRINT()	Debugging Tools
A partition is too small for the	Error Codes
B_PARTITION_TOO_SMAL	Error Codes
B_PATH_TYPE	Constants and Defined Types
B_PATTERN_TYPE	Constants and Defined Types
B_PERMISSION_DENIED	Error Codes
B_POINT_TYPE	Constants and Defined Types
B_POINTER_TYPE	Constants and Defined Types
Position()	BMallocIO

Position()	BMemoryIO
Position()	BPositionIO
BPositionIO	BPositionIO
BPositionIO()	BPositionIO
~BPositionIO()	BPositionIO
Prepend()	BString
PRINT_OBJECT()	Debugging Tools

R

B_RAW_TYPE	Constants and Defined Types
ReadAt()	BDataIO
ReadAt()	BMallocIO
ReadAt()	BMemoryIO
ReadAt()	BPositionIO
Read()	BDataIO
Read()	BPositionIO
B_READ_ONLY_DEVICE	Error Codes
B_RECT_TYPE	Constants and Defined Types
B_REF_TYPE	Constants and Defined Types
RemoveAll()	BString
Remove()	BString
RemoveFirst()	BString
RemoveItem()	BList

RemoveItems()	BList
RemoveLast()	BString
RemoveSet()	BString
Replace()	BString
ReplaceFirst()	BString
ReplaceLast()	BString
ReplaceSet()	BString
The requested (unique) name is already used	Error Codes
Reset()	BStopWatch
Resume()	BStopWatch
B_RGB_32_BIT_TYPE	Constants and Defined Types
B_RGB_COLOR_TYPE	Constants and Defined Types

S

ASSERT()	Debugging Tools
Same as	Error Codes
Save()	BBlockCache
Seek()	BMallocIO
Seek()	BMemoryIO
Seek()	BPositionIO
Sem()	BLocker
SERIAL_PRINT()	Debugging Tools
SERIAL_TRACE()	Debugging Tools

SetBlockSize()	BMallocIO
SetCount()	BMessageRunner
SetInterval()	BMessageRunner
SetSize()	BMallocIO
SetSize()	BPositionIO
SetTo()	BString
SET_DEBUG_ENABLED()	Debugging Tools
B_SIZE_T_TYPE	Constants and Defined Types
B_SJIS_CONVERSION	Constants and Defined Types
SortItems()	BList
B_SSIZE_T_TYPE	Constants and Defined Types
Static Functions	BArchivable
status_t	Constants and Defined Types
BStopWatch	BStopWatch
BStopWatch()	BStopWatch
~BStopWatch()	BStopWatch
BString	BString
String()	BString
~BString()	BString
B_STRING_TYPE	Constants and Defined Types
The Support Kit	The Support Kit
The Support Kit	The Support Kit
Suspend()	BStopWatch
B_SWAP_ALWAYS	Constants and Defined Types
B_SWAP_BENDIAN_TO_HOST	Constants and Defined Types
swap_data()	Functions and Macros

B_SWAP_HOST_TO_BENDIAN	Constants and Defined Types
B_SWAP_HOST_TO_LENDIAN	Constants and Defined Types
B_SWAP_LENDIAN_TO_HOST	Constants and Defined Types
system_beep()	Functions and Macros

T

TRACE()	Debugging Tools
TRESPASS()	Debugging Tools
There's no match for the specified name.	Error Codes
Thread Local Storage Functions	Thread Local Storage
Thread Local Storage	Thread Local Storage
Thread Local Storage	Thread Local Storage
B_TIME_TYPE	Constants and Defined Types
B_TIMED_OUT	Error Codes
tls_address()	Thread Local Storage
tls_allocate()	Thread Local Storage
tls_get()	Thread Local Storage
tls_set()	Thread Local Storage
ToLower()	BString
ToUpper()	BString
true	Constants and Defined Types
Truncate()	BString
TypeCode()	BFlattenable

Type Codes	Constants and Defined Types
type_code	Constants and Defined Types

U

B_UNSUPPORTED	Error Codes
URL MIME Types	Constants and Defined Types
B_UINT16_TYPE	Constants and Defined Types
B_UINT32_TYPE	Constants and Defined Types
B_UINT64_TYPE	Constants and Defined Types
B_UINT8_TYPE	Constants and Defined Types
Unarchiving	Archiving
Unflatten()	BFlattenable
B_UNKOWN_MIME_TYPE	Error Codes
Unlock()	BLocker
UnlockBuffer()	BString
B_URL_FILE	Constants and Defined Types
B_URL_FTP	Constants and Defined Types
B_URL_GOPHER	Constants and Defined Types
B_URL_HTTP	Constants and Defined Types
B_URL_HTTPS	Constants and Defined Types
B_URL_MAILTO	Constants and Defined Types
B_URL_NEWS	Constants and Defined Types
B_URL_NNTP	Constants and Defined Types

B_URL_RLOGIN	Constants and Defined Types
B_URL_TELNET	Constants and Defined Types
B_URL_TN3270	Constants and Defined Types
B_URL_WAIS	Constants and Defined Types

V

validate_instantiation()	Functions and Macros

W

B_WOULD_BLOCK	Error Codes
WriteAt()	BMallocIO
WriteAt()	BMemoryIO
WriteAt()	BPositionIO
Write()	BDataIO
Write()	BPositionIO

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BString

Derived from: (none)
Declared in: be/support/String.h
Library: libbe.so
Allocation: Constructor or on the stack
Summary: *more...*

BString is a string allocation and manipulation class. The object takes care of allocating and freeing memory for you, and provides a number of character search, comparison, and manipulation functions in a variety of flavors: `FindFirst()`, `FindLast()`, `Prepend()`, `Append()`, `Insert()`, `Remove()`, `RemoveSet()`, and so on. The class also defines a number of operators that map to these functions; for example, the `+=` operator is the same as the `Append()` function. The operators make BString objects particularly easy to work with.

This document describes some global C functions and operators that act like BString functions. The global `Compare()` and `ICompare()` functions let you compare two BStrings that are passed as arguments; they're meant to be used as hook functions that are called from within a sorting routine (such as **`qsort()`**). The global operators are defined so you don't have to worry about the order of the operands.

BString is not thread safe. You shouldn't try to access the same BString object from two different threads at the same time.

Currently, BString only supports assignment and retrieval of strings that contain single-byte characters. If you want to use a BString to store a string that contains multi-byte (UTF8) characters, you have to flatten the string yourself and adjust the character count arguments accordingly.

Assigning and Retrieving String Data

To assign a BString's string, you pass the string to the object's constructor, to **SetTo()**, or you use the = or << operators:

```
BString string("8 Bits");

/* or */
BString string;
string.SetTo("8 Bits");

/* or */
BString string = "8 Bits";

/* or */
BString string;
string << (int32)8 << " Bits";
```

There's also an **Adopt()** function that let's you move data from one BString object to another. In all cases, the BString object takes care of allocating storage for you—the object is guaranteed to be "big enough" to contain the string data.

The **String()** function retrieves the object's string:

```
myButton->SetLabel(string.String());
```

ByteAt() returns a single character, located by index. You can also get a character by using the `[]` operator; the following returns the string's first character:

```
char c = string[0];
```

Direct Data Access

If you want to access a BString's data directly, you call **LockBuffer()**, and then call **UnlockBuffer()** when you're finished. Between these two calls, you can manipulate the string data, but you mustn't call any other BString function:

```
int32 len = string.Length()+1
char *stringData = string.LockBuffer(len);

/* While the buffer is locked you can manipulate the string by hand, or through standard C fun
but you mustn't call BString functions. */

stringData[0] = 'd';
strcat(stringData, " button");
string.UnlockBuffer(len+strlen(" button"));

/* The object can once again receive BString functions. */
```

Constructor and Destructor

BString()

```
BString(void)
BString(const char *string)
BString(const char *string, int32 maxLength)
BString(const BString &string)
```

Creates a new BString object that allocates enough storage to accommodate *string* (or *string*->`String()`, or up to *maxLength* bytes), and then copies the string into the storage. Without an argument, the new BString is empty. You can also set a BString's data by using `SetTo()`, `Adopt()`, or the `=` operator.

~BString()

```
~BString()
```

Frees the object's allocated memory, and destroys the object.

Member Functions

- Adopt() see SetTo()

- Append() , Prepend() , Insert()

inline BString &**Append**(const BString &*source*) BString &**Append**(const BString &*source*, int32 *charCount*) inline BString &**Append**(const char **source*) BString &**Append**(const char **source*, int32 *charCount*) BString &**Append**(char *c*, int32 *charCount*)
BString &**Prepend**(const BString &*source*) BString &**Prepend**(const BString &*source*, int32 *charCount*) BString &**Prepend**(const char **source*) BString &**Prepend**(const char **source*, int32 *charCount*) BString &**Prepend**(char *c*, int32 *charCount*)
BString &**Insert**(const BString &*source*, int32 *insertAt*) BString &**Insert**(const BString &*source*, int32 *charCount*, int32 *insertAt*) BString &**Insert**(const BString &*source*, int32 *sourceOffset*, int32 *charCount*, int32 *insertAt*) BString &**Insert**(const char **source*, int32 *insertAt*) BString &**Insert**(const char **source*, int32 *charCount*, int32 *insertAt*) BString &**Insert**(const char **source*, int32 *sourceOffset*, int32 *charCount*, int32 *insertAt*) BString &**Insert**(char *c*, int32 *charCount*, int32 *insertAt*)

These functions add characters to the end (append), beginning (prepend), or middle (insert) of the BString's string. In each case, the BString automatically reallocates to accommodate the new data. All of these functions return ***this**.

Append() copies *charCount* characters from *source* and adds them to the end of this BString. If *charCount* isn't specified, the entire string is copied; this is the same as the **+=** operator. The single character version of **Append()** adds *count* copies of the character *c* to the end of the object.

Prepend() does the same as **Append()**, except it adds the characters to the beginning of this BString, shifting the existing data "to the right" to make room.

Insert() adds the designated characters at *insertAt* (zero-based) in this BString. The BString's existing data (at location *insertAt* and higher) is shifted right to make room for the new data. The *sourceOffset* argument is an offset (zero-based) into the source string.

- ByteAt() see String()
- CharacterDeescape() see CharacterEscape()

CharacterEscape() , CharacterDeescape()

```
BString &CharacterEscape(const char *original,
    const char *setOfCharsToEscape, 1w
    char escapeWithChar)
BString &CharacterEscape(const char *setOfCharsToEscape, char escapeWithChar)
```

```
BString &CharacterDeescape(const char *original, char escapeWithChar)
BString &CharacterDeescape(char escapeWithChar)
```

CharacterEscape() scans a string for occurrences of the individual characters in the string *setOfCharsToEscape* and inserts into the string the byte indicated by *escapeWithChar* before each such occurrence. The first form copies the original string into the BString; the second form operates on the existing BString.

CharacterDeescape() undoes this operation by scanning the string and removing all occurrences of the byte *escapeWithChar*. The first form copies the *original* string into the BString, while the second form operates on the existing BString.

Compare() , ICompare()

```
int Compare(const BString &string) const
int Compare(const BString &string, int32 range) const
int Compare(const char *string) const
int Compare(const char *string, int32 range) const
int Compare(const BString &astring, const BString &bstring) global
int Compare(const BString *astring, const BString *bstring) global
```

```
int ICompare(const BString &string) const
int ICompare(const BString &string, int32 range) const
int ICompare(const char *string) const
int ICompare(const char *string, int32 range) const
int ICompare(const BString &astring, const BString &bstring) global
int ICompare(const BString *astring, const BString *bstring) global
```

These functions compare this BString with the argument string (Compare() is case-sensitive, ICompare() is case-insensitive); they return 0 if the two strings are the same, 1 if this BString is "greater than" the argument, and -1 if the argument is greater than this BString. Two strings are compared by comparing the ASCII or UTF-8 values of their respective characters. A longer string is greater than a shorter (but otherwise similar) string—"abcdef" is greater than "abcde".

Given a *range* argument, the functions only compare the first *range* characters.

The global functions return 0 if the arguments are equal, 1 if *astring* is greater than *bstring*, and -1 if *bstring* is greater than *astring*.

You can also compare two strings through the comparison operators, ==, !=, <, <=, >, and =>.

CopyInto() , MoveInto()

```
BString &CopyInto(BString &destination, int32 sourceOffset,
    int32 charCount) const
void CopyInto(char *destination, int32 sourceOffset, int32 charCount) const
```

```
BString &MoveInto(BString &destination, int32 sourceOffset, int32 charCount)
void MoveInto(char *destination, int32 sourceOffset, int32 charCount)
```

`CopyInto()` copies a substring from this BString into *destination*. `MoveTo()` does the same, but it removes the original substring (from this BString). If *destination* is a BString, storage for the substring is automatically allocated; if the destination is a **`char *`**, the caller is responsible for allocating sufficient storage.

The substring comprises *charCount* characters starting at character *sourceOffset* (zero-based). After the substring is removed, the remaining characters move (to the left) to fill in the gap (`MoveTo()` only). For example:

```
BString source, destination;
source.SetTo("abcdefg");
source.MoveInto(&destination, 2, 3);

/* source.String() == "abfg" */
/* destination.String() == "cde" */
```

The functions return *destination* (or void).

CountChars() , Length()

```
int32 CountChars(void) const
```

```
inline int32 Length(void) const
```

`CountChars()` returns the length of the object's string measured in characters; `Length()` returns the length measured in bytes.

FindFirst() , IFindFirst() , FindLast() , IFindLast()

```
int32 FindFirst(const BString &string) const
int32 FindFirst(const BString &string, int32 offset) const
int32 FindFirst(const char *string) const
int32 FindFirst(const char *string, int32 offset) const
int32 FindFirst(char c) const
int32 FindFirst(char c, int32 offset) const
```

```
int32 FindLast(const BString &string) const
int32 FindLast(const BString &string, int32 offset) const
```

int32 **FindLast(**const char *_string_**)** const
int32 **FindLast(**const char *_string_, int32 _offset_**)** const
int32 **FindLast(**char _c_**)** const
int32 **FindLast(**char _c_, int32 _offset_**)** const

int32 **IFindFirst(**const BString &_string_**)** const
int32 **IFindFirst(**const BString &_string_, int32 _offset_**)** const
int32 **IFindFirst(**const char *_string_**)** const
int32 **IFindFirst(**const char *_string_, int32 _offset_**)** const

int32 **IFindLast(**const BString &_string_**)** const
int32 **IFindLast(**const BString &_string_, int32 _offset_**)** const
int32 **IFindLast(**const char *_string_**)** const
int32 **IFindLast(**const char *_string_, int32 _offset_**)** const

These functions return the index of the first or last occurrence (within this BString) of a substring or character. **FindFirst()** and **FindLast()** are case-sensitive; **IFindFirst()** and **IFindLast()** are case-insensitive. The functions return **B_ERROR** if the character isn't found.

The *offset* versions only look in the portion of this BString that starts at character *offset* (for [**I**]**FindFirst()**), or that ends at character *offset* (for [**I**]**FindLast()**). For example, in this example...

```
BString astring("AbcAbcAbc");
astring.FindLast("Abc", 7);
```

...the **FindLast()** call returns 3, the index of the last complete instance of "Abc" that occurs before character 7.

- I see Compare()
- Insert() see Append()
- Length() see CountChars()

- LockBuffer() , UnlockBuffer()

char ***LockBuffer(**int32 _maxLength_**)**
BString &**UnlockBuffer(**int32 _length_ = -1**)**

LockBuffer() returns a pointer to the object's string; you're allowed to manipulate this pointer directly. The *maxLength* argument lets you ask the object to allocate some extra space at the end of the string before passing you the pointer. If *maxLength* is less than the string's current length, the argument is ignored (pass 0 if you want to lock the buffer but don't need to pre-allocate extra space).

UnlockBuffer() tells the object that you're done manipulating the string pointer. *length* is the string's new length; if you pass -1, the BString gets the length by calling **strlen()** on the string.

The functions returns *this.

Every LockBuffer() must be balanced by a subsequent UnlockBuffer(). In between the two calls, you may not call any BString functions. Many of the BString functions assert an error if they're called while the buffer is locked.

See "Direct Data Access" for an example.

- MoveInto() see CopyInto()
- Prepend() see Append()

- Remove() , RemoveFirst() , RemoveLast() , RemoveAll() , RemoveSet()

BString &**Remove(**int32 *startingAt*, int32 *charCount***)**
BString &**RemoveFirst(**const BString &*string***)** BString &**RemoveFirst(**const char **string***)**
BString &**RemoveLast(**const BString &*string***)** BString &**RemoveLast(**const char **string***)**
BString &**RemoveAll(**const BString &*string***)** BString &**RemoveAll(**const char **string***)**
BString &**RemoveSet(**const char **charSet***)**

These functions remove characters from the BString and reallocate the object's storage so it fits the new (smaller) data. The functions return *this.

Remove() removes *charCount* characters, starting with character *startingAt* (zero-based).

RemoveFirst(), RemoveLast(), and RemoveAll() remove, respectively, the first, last and every occurrence of *string* within the object.

RemoveSet() removes all occurrences of every character in the *charSet* string. For example:

```
BString string("ab abc abcd");
string.RemoveSet("db");
/* 'string' now contains "a ac ac" */
```

- Replace() , ReplaceFirst() , ReplaceLast() , ReplaceSet() , IReplace() , IReplaceFirst() , IReplaceLast()

BString &**Replace(**const char **old*, const char **new*, int32 *count*, int32 *offset* = 0**)** BString &**Replace(**char *old*, char *new*, int32 *count*, int32 *offset* = 0**)**
BString &**ReplaceFirst(**const char **old*, const char **new***)** BString &**ReplaceFirst(**char *old*, char *new***)**
BString &**ReplaceLast(**const char **old*, const char **new***)**

BString &**ReplaceLast(**char *old*, char *new***)**

BString &**ReplaceAll(**const char **old*, const char **new*, int32 *offset* = 0**)**
BString &**ReplaceAll(**char *old*, char *new*, int32 *offset* = 0**)**

BString &**ReplaceSet(**const char **oldSet*, const char **new***)**
BString &**ReplaceSet(**const char **oldSet*, char *new***)**

BString &**IReplace(**const char **old*, const char **new*, int32 *count*, int32 *offset* = 0**)**
BString &**IReplace(**char *old*, char *new*, int32 *count*, int32 *offset* = 0**)**

BString &**IReplaceFirst(**const char **old*, const char **new***)**
BString &**IReplaceFirst(**char *old*, char *new***)**

BString &**IReplaceLast(**const char **old*, const char **new***)**
BString &**IReplaceLast(**char *old*, char *new***)**

BString &**IReplaceAll(**const char **old*, const char **new*, int32 *offset* = 0**)**
BString &**IReplaceAll(**char *old*, char *new*, int32 *offset* = 0**)**

These functions find occurrences of *old* within the BString, and replace them with *new*. In the **Replace...()** functions, the search for *old* is case-insensitive; in the **IReplace...()** functions, the search is case-insensitive. The functions return ***this**.

[I]Replace() replaces the first *count* occurrences that start on or after the character at *offset*.

[I]ReplaceFirst() and **[I]ReplaceLast()** replace only the first and last occurrence (respectively).

[I]ReplaceAll() replaces all occurrence that start on or after the character at *offset*.

[I]ReplaceSet() is slightly different from the others: It replaces *each* occurrence of *any* character in *oldSet* with the character or the entire string given by *new*. For example:

```
BString astring("a-b-c");
astring.ReplaceSet("abc", "ABC");
/* astring is now "ABC-ABC-ABC" */
```

SetTo() , Adopt()

inline BString &**SetTo(**const char **source***)**
BString &**SetTo(**const char **source*, int32 *charCount***)**
BString &**SetTo(**const BString &*source***)**
BString &**SetTo(**const BString &*source*, int32 *charCount***)**
BString &**SetTo(**char *c*, int32 *charCount***)**

BString &**Adopt(**BString &*source***)**
BString &**Adopt(**BString &*source*, int32 *charCount***)**

These functions initialize a BString's string data. Storage for the data is automatically allocated by the functions. The object's current data is wholly replaced by the new data. The functions return ***this**.

SetTo() copies *charCount* characters from *source* into this object. If *charCount* isn't given, the entire string is copied; this form is equivalent to the **=** operator. The single character version of **SetTo()** sets the object's data to a *charCount*-length string that consists entirely of the character *c*.

Adopt() moves *charCount* characters from *source* into this object, and then clears *source*'s data (pointer). Note that *source* will be empty after you call **Adopt()**, even if *charCount* is less than *source*'s full length.

String() , ByteAt()

inline const char ***String()** const
inline char **ByteAt(**int32 *index***)** const

String() returns a pointer to the object's string, guaranteed to be **NULL** terminated. You may not modify or free the pointer. If the BString is deleted, the pointer becomes invalid.

ByteAt() returns the *index*'th character in the string. If *index* is out of bounds, the function returns 0. Except for the boundary check, this function is the same as the **[]** operator.

ToLower() , ToUpper() , Capitalize() , CapitalizeEachWord()

BString &**ToLower(**void**)**
BString &**ToUpper(**void**)**
BString &**Capitalize(**void**)**
BString &**CapitalizeEachWord(**void**)**

ToLower() converts every character in the string to lower case; **ToUpper()** converts them all to upper case. Non-alphabetic characters aren't affected.

Capitalize() converts the first alphabetic character in the word to upper case, and the rest to lower case. **CapitalizeEachWord()** converts the first alphabetic character in each "word" to upper case, and the rest to lower case. A word is a group of alphabetic characters delimited by non-alphabetic characters.

The functions return ***this**.

Capitalize() and **CapitalizeEachWord()** are broken in Release 4.0.

Truncate()

BString &**Truncate(**int32 *charCount*, bool *lazy* = **true)**

Truncate() shrinks the object's string so that it's *charCount* characters long. This function will not lengthen the string: If *charCount* is equal to or greater than the string's current character count, the function does nothing.

If *lazy* is **false**, the string is immediately reallocated to trim it to the new size (and the excess is immediately freed). If it's **true**, the string is set to the new size, but the excess isn't freed until the next storage manipulating function is called. It's slightly more efficient to be lazy; otherwise, the two forms of the function are identical.

The function returns ***this**.

LockBuffer()

Operators

= (assignment)

```
BString& operator=(const BString &string)
BString& operator=(const char *string)
BString& operator=(const char character)
```

Sets the contents of the BString to *string* or *character*, and returns ***this**LockBuffer()_.

+= (append)

```
inline BString& operator+=(const BString &string)
BString& operator+=(const char *string)
BString& operator+=(const char character)
```

Appends *string* or *character* to this BString, and returns ***this**.

<< (formatted append)

```
BString &operator<<(const char *string)
BString &operator<<(const BString &string)
BString &operator<<(char c)
BString &operator<<(uint32 val)
BString &operator<<(int32 val)
BString &operator<<(uint64 val)
BString &operator<<(int64 val)
BString &operator<<(float val)
```

Converts the right operand to a string (if necessary), and appends it to the left operand. Floating point values are always written with two decimal digits:

```
BString astring("The result is: "), bstring(" bps");
float result = 12.5;
astring << result << bstring;
```

Here, *astring* becomes "The result is: 12.50 bps".

Multiple append operations are evaluated from left to right, so that only the leftmost operand is modified. For example:

```
BString astring("a"), bstring("b"), cstring("c");
astring << bstring << cstring;
```

Here, *bstring* is appended to *astring*, and then *cstring* is appended to the result; thus, *astring* is "abc", and *bstring* is still "b".

[] (indexing)

```
char operator[](int32 index) const
inline char &operator[](int32 index)
```

Returns the character at *index* in the string. No boundary checking is done –– it's up to the caller to ensure that *index* is in bounds.

== , != , < , > , <= , >= (comparison)

```
inline bool operator==(const BString &string) const
bool operator==(const char *string) const
global inline bool operator==(const char *string, const BString &string) const
```

```
inline bool operator!=(const BString &string) const
inline bool operator!=(const char *string) const
global inline bool operator!=(const char *string, const BString &string) const
```

```
inline bool operator<(const BString &string) const
bool operator<(const char *string) const
global inline bool operator<(const char *string, const BString &string) const
```

```
inline bool operator<=(const BString &string) const
bool operator<=(const char *string) const
global inline bool operator<=(const char *string, const BString &string) const
```

```
inline bool operator>(const BString &string) const
bool operator>(const char *string) const
global inline bool operator>(const char *string, const BString &string) const
```

```
inline bool operator>=(const BString &string) const
bool operator>=(const char *string) const
global inline bool operator>=(const char *string, const BString &string) const
```

Case-sensitive comparison of two strings. Two strings are compared by comparing the ASCII or UTF-8 values of their respective characters. A longer string is greater than a shorter (but otherwise similar) string—"abcdef" is greater than "abcde".

The global versions of these operators are provided so you don't have to worry about the order of the operands; for example:

```
if (astring == "Okay")
/* ...is equivalent to... */
if ("Okay" == astring)
```

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BString Function Summary

Function Summary

Function	Description
!=	Case-sensitive comparison of two strings
>	Case-sensitive comparison of two strings
>=	Case-sensitive comparison of two strings
<	Case-sensitive comparison of two strings
<<	Appends the formatted representation of a string or number to the object
<=	Case-sensitive comparison of two strings
+=	Appends a string or character to the BString
=	Copies the contents of one string or character to the BString
==	Case-sensitive comparison of two strings
Adopt()	Moves string data from the argument into this BString.
Append()	Adds characters to the end of the BString.
ByteAt()	Returns the character at a given

	position in the string
Capitalize()	Converts the string's first character to upper case, the rest to lower
CapitalizeEachWord()	Converts the first letter of each word to upper case, the rest to lower
CharacterDeescape()	Removes escape characters from a string
CharacterEscape()	Escapes specified characters by inserting a special character in front of them
Compare()	Case-sensitive comparison
CopyInto()	Copies the string into another object or buffer.
CountChars()	Returns the string's character count.
FindFirst()	Returns the index of the first occurrence of a letter or string
FindLast()	Returns the index of the first occurrence of a letter or string, case-insensitive
ICompare()	Case-insensitive comparison
IFindFirst()	Returns the index of the last occurrence of a letter or string
IFindLast()	Returns the index of the last occurrence of a letter or string, case-insensitive
IReplace()	Replaces all occurrences of a letter or string with another letter or string
IReplaceFirst()	Replaces a case-insensitive string

	with another string
IReplaceLast()	Returns the index of the first occurrence of a letter or string, case-insensitive
Insert()	Adds characters to the middle of the BString.
Length()	Returns the string's length in bytes.
LockBuffer()	Gives direct access to the object's string
MoveInto()	Moves the string into another object or buffer.
Prepend()	Adds characters to the beginning of the BString.
RemoveAll()	Removes every occurrence of a string within the BString
Remove()	Removes some characters from the BString
RemoveFirst()	Removes the first occurrence of a string within the BString
RemoveLast()	Removes the last occurrence of a string within the BString
RemoveSet()	Removes every occurrence of a set of characters within the BString.
Replace()	Replaces a letter or string with another letter or string
ReplaceFirst()	Replaces the first occurrence of a letter or string with another letter or string

ReplaceLast()	Replaces the last occurrence of a letter or string with another letter or string
ReplaceSet()	Replaces all occurrences of a letter or string with another letter or string
SetTo()	Copies string data from the argument into this BString.
String()	Returns the object's string
ToLower()	Converts the string to lower case
ToUpper()	Converts the string to lower case
Truncate()	Sets the size of the string.
UnlockBuffer()	Balances the previous LockBuffer() call

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaAddOn

Derived from: none
Declared in: be/media/MediaAddOn.h **Library:** libmedia.so
Allocation: Constructor only
Summary: *more...*

A BMediaAddOn-derived object describes an executable program that lives in a disk file and is loaded by the Media Server when it's needed. A BMediaAddOn tells the Media Server what kinds of nodes it can create and handles the actual creation of those nodes when called upon by the Media Server to do so.

It's important to note that the functions in the BMediaAddOn class will typically only be called by the Media Kit (and from within the add-on itself). These functions aren't called by client applications.

Getting to Node You...

A given node can support as many media kinds and formats as it wants (although if you support too many widely disparate media types, your add-on may get difficult to maintain, but that's another issue entirely). For example, a node that supports video might support inputting AVI, QuickTime, and MPEG-2, but might only be able to output AVI. This is information that the Media Kit needs to know. For this reason, the BMediaAddOn needs to provide information about the media **flavors** it supports.

This is done using the flavor_info structure:

```
struct flavor_info {
   char *name;
   char *info;
   uint64 kinds;
   uint32 flavor_flags;
   int32 internal_id;
   int32 possible_count;

   int32 in_format_count;
   uint32 in_format_flags;
   const media_format *in_formats;

   int32 out_format_count;
   uint32 out_format_flags;
   const media_format *out_formats;

   uint32 _reserved_[16];
```

```
private:
   flavor_info & operator=(const flavor_info &other);
};
```

The *name* and *info* fields provide a human-readable name, and information about the flavor.

The *kinds* field should indicate all the relevant kinds that the node matches; this is a bit field, and it's possible that more than one flag may be relevant. See "node_kind" on page202.

The *flavor_flags* field contains flags providing additional information about the flavor.

Constant	Description
B_FLAVOR_IS_GLOBAL	The flavor will be forced into the Media Add-on Server, and only one instance of it will exist.
B_FLAVOR_IS_LOCAL	The flavor will be forced into the loading application, and many instances of it may exist.

If neither flag is specified, the Media Kit will decide what to do with the flavor.

The *internal_id* is an internal ID number that your add-on can use to identify the flavor; the flavor will be requested by the Media Kit using this ID number.

The *possible_count* field specifies to the Media Kit the maximum number of instances of your node can be in existence at the same time. For example, if your node provides support for a particular sound card, this value should be equal to the number of cards you support that are currently installed in the computer.

in_format_count specifies how many input formats the flavor supports, and *in_formats* is a list of all the input formats supported by the flavor.

in_format_flags provides informational flags about the flavor's inputs. There aren't any defined values for this field yet; be sure to set it to 0.

out_format_count specifies how many output formats the flavor supports, and *out_formats* is a list of all the output formats supported by the flavor.

out_format_flags provides informational flags about the flavor's outputs. There aren't any defined values for this field yet; be sure to set it to 0.

If your node is a physical input, such as a sound card, your node's *kinds* field should include **B_PHYSICAL_INPUT** among the flags set therein. Likewise, if your node is a physical output, or a system mixer, you should include **B_PHYSICAL_OUTPUT** or **B_SYSTEM_MIXER**.

Your node's constructor should also call **AddNodeKind()** to add these kind flags; the base classes only add **B_BUFFER_CONSUMER**, **B_BUFFER_PRODUCER**, and so forth; the flags indicating that the node represents a physical input, physical output, or system mixer aren't added automatically. For example, a sound digitizer node's constructor might have the following form:

```
MyBufferProducer::MyBufferProducer(const char *name) :
         BMediaNode(name),
         BBufferProducer() {

   AddNodeKind(B_PHYSICAL_INPUT);

   /* constructor stuff goes here */
}
```

Hook Functions

- AutoStart()
- CountFlavors()
- GetConfigurationFor()
- GetFileFormatList()
- GetFlavorAt()
- InitCheck()
- InstantiateNodeFor()
- SniffRef()
- SniffType()
- SniffTypeKind()
- WantsAutoStart()

Constructor and Destructor

BMediaAddOn()

```
explicit BMediaAddOn(image_id image)
```

This is the BMediaAddOn constructor; your derived class should perform whatever initialization is necessary in the constructor, and handle any activities necessary in order to properly comply with the media add-on protocol.

The image_id identifies the add-on image from which the add-on was loaded. Your `make_media_addon()` function should be sure to save the image ID passed to it and pass that value through to the BMediaAddOn constructor, or the Media Kit will be unhappy with you.

~BMediaAddOn

```
~BMediaAddOn()
```

You should never delete a BMediaAddOn yourself; the Media Server will do this when the add-on is no longer needed. However, if you're implementing a media add-on, this is a good place to handle disposing of memory you've allocated.

Member Functions

AutoStart()

```
virtual status_t AutoStart(int32 index, BMediaNode **outNode,
    int32 *outInternalID, bool *outHasMore) = 0
```

If `WantsAutoStart()` returns **true**, you'll get repeated calls to `AutoStart()` when your add-on is loaded.

Your `AutoStart()` function should be implemented to instantiate nodes to your liking, one for each call to `AutoStart()`. Each time `AutoStart()` is called, *index* will be one higher than the previous pass. After creating the new node, store a pointer to it in *outNode*, and set *outInternalID* to the internal ID of the flavor represented by that node.

Before returning, set *outHasMore* to true if there are more nodes to start, and false if there aren't.

Return `B_OK` if you return a valid node. If there aren't any more nodes to be started, return `B_BAD_INDEX`. If the specified index fails, but there may be other nodes to start, return **B_ADDON_FAILED**.

CountFlavors()

```
virtual int32 CountFlavors(void)
```

Implement this hook function to return the number of flavors your add-on can deal with.

GetConfigurationFor()

```
virtual status_t GetConfigurationFor(BMediaNode *node, BMessage *config)
```

Implement this function to save information about how the given node is configured into the specified message. This lets the node's configuration be saved to disk for future rehydration using `InstantiateNodeFor()`.

Note that it's not reasonable to save information about what connections are routed through the node, since the other nodes probably won't exist when the node is rehydrated. It's up to the application to reconstruct the connections itself, as it wishes.

If the configuration information is stashed into *config* without errors, return `B_OK`; otherwise, return an appropriate error code.

GetFileFormatList()

```
virtual status_t GetFileFormatList(int32 flavorID,
    media_file_format *outWritableFormats,
    int32 inWriteItems,
    int32 *outWriteItems,
    media_file_format *outReadableFormats,
    int32 inReadItems,
    int32 *outReadItems,
    void *reserved)
```

If your add-on provides nodes that can either read or write media data from files, you should implement this function to describe the `media_file_formats` you can read and write.

On input, the *flavorID* argument indicates which flavor you should return information about. You should fill the array *outWritableFormats* with a list of the file formats the corresponding flavor can write, and the *outReadableFormats* array with a list of the file formats the flavor can read. The *inWriteItems* and *inReadItems* arguments indicate how many items can be stored in each array.

Both *outWritableFormats* and *outReadableFormats* can be **NULL**. Don't write into these arrays if the corresponding pointer is **NULL**.

On output, you should set *outWriteItems* to be the number of file formats you can write—even if it's larger than *inWriteItems*. This lets the caller resize the array and call you back to get the complete list. Likewise, *outReadItems* should be set to the number of file formats you can read.

The *reserved* argument isn't currently used, and should always be specified as **NULL**.

Only implement this function if your node derives from BFileInterface.

If an error occurs, return an appropriate error code; otherwise, return **B_OK**.

GetFlavorAt()

```
virtual status_t GetFlavorAt(int32 flavorNum, const flavor_info **outInfo)
```

Implement `GetFlavorAt()` to store in *outInfo* a pointer to a static **flavor_info** structure describing one flavor supported by your add-on. The data pointed to by *outInfo* will be copied by the Media Kit after `GetFlavorAt()` returns. For example, if your add-on only supports one flavor:

```
flavor_info flavorInfo;

status_t MyMediaAddOn::GetFlavorAt(int32 n, const flavor_info **out_info) {
   if (n != 0) {
```

```
      return B_ERROR;
   }

   flavorInfo->internal_id = n;
   flavorInfo->name = strdup("My Media Node");
   flavorInfo->info = strdup("This node receives video from spy cameras at area pizza parlors
determines which one is the least busy.");
   flavorInfo->kinds = B_BUFFER_CONSUMER | B_PHYSICAL_OUTPUT;
   flavorInfo->flavor_flags = 0;
   flavorInfo->possible_count = 0;

   // Set up the list of input formats. We only support
   // raw video input.

   flavorInfo->in_format_count = 1;
   media_format *aFormat = new media_format;
   aFormat->type = B_MEDIA_RAW_VIDEO;
   aFormat->u.raw_video = media_raw_video_format::wildcard;
   flavorInfo->in_formats = aFormat;

   // We don't output.

   flavorInfo->out_format_count = 0;
   flavorInfo->out_formats = 0;

   // And set up the result pointer

   *out_info = flavorInfo;
   return B_OK;
}
```

If you allocate memory for the **flavor_info** structure, be sure to discard it before your add-on is unloaded (the destructor is a good place to do this). If you forget to discard the memory, you'll leak, and that can be really embarrassing.

If *flavorNum* is greater than **CountFlavors()**-1 (in other words, if the requested number is larger than the number of flavors you support), return **B_ERROR**. Otherwise, return **B_OK** or, if an error occurs, an appropriate error code.

ImageID()

image_id **ImageID**(void)

Returns the add-on's image ID, as specified by the BMediaAddOn constructor.

Your **make_media_addon()** function should be sure to save the image ID passed to it and pass that value through to the BMediaAddOn constructor, or the Media Kit will be unhappy with you.

InitCheck()

virtual status_t **InitCheck(**const char ***outFailureText*)**

Implement this hook function to set *outFailureText* to point to a static text buffer that contains a message explaining why your add-on failed the most recent operation requested by the Media Server. This doesn't include errors reported by nodes instantiated by your add-on, but only to errors caused directly by your BMediaAddOn-derived object.

The buffer pointed to by *outFailureText* shouldn't go away until the add-on is reloaded or a new call to `InitCheck()` is made.

Also returned is the actual result code from that operation; if all is well, **B_OK** should be returned.

InstantiateNodeFor()

virtual BMediaNode ***InstantiateNodeFor(**const flavor_info ****info***,
BMessage ****config***, status_t ****outError***)**

Given the information in *info* (which might be a copy of information you previously returned from a **GetFlavorAt()** call), you should instantiate a new object of the node class referenced by *info*'s internal_id field and return a pointer to that node.

The *config* message might contain information saved by a previous instance of the same flavor node, and should be used to configure the new node if possible. *config* will never be **NULL**, although it may not have any fields of interest in it.

The simplest implementation of this function, in which there's no configuration possible, might look like this:

```
BMediaNode *BMyConsumerAddOn::InstantiateNodeFor(const flavor_info *info,
            BMessage *config, status_t *outError) {
   return new MyConsumerNode((const char *) "My Consumer Node");
}
```

If a new instance can't be instantiated, set *outError* to an appropriate error code and return **NULL**. Otherwise, set *outError* to **B_OK**.

SniffRef() , SniffType() , SniffTypeKind()

```
virtual status_t SniffRef(const entry_ref &file, BMimeType *outMimeType,
    float *outQuality, int32 *outInternalID)
```

```
virtual status_t SniffType(BMimeType &mimeType,
    float *outQuality, int32 *outInternalID)
```

```
virtual status_t SniffTypeKind(BMimeType &mimeType, uint64 inKinds,
    float *outQuality, int32 *outInternalID,
    void *reserved)
```

If your add-on supports BFileInterface nodes, you may implement `SniffRef()`. Given the specified *file*, examine the file and set *outMimeType* to describe the MIME type of the file.

Additionally, you can implement `SniffType()` to report how well your add-on can handle a file containing data of the MIME type specified by *mimeType*.

If your node deals with both producers and consumers, you shouldn't implement `SniffType()`. Implement **`SniffTypeKind()`** instead. If **`SniffTypeKind()`** is implemented, `SniffType()` won't be called.

`SniffTypeKind()` limits the search to the kinds indicated by *inKinds*. This lets the caller indicate whether it's looking for a consumer or a producer, for example.

In any case, you should also store, in *outQuality*, a value representing the quality level at which your add-on can interpret the data in the file (where 0.0 means you can't handle it well at all and 1.0 means you can handle it perfectly), and in *outInternalID*, the internal ID of the flavor that handles the data.

Don't implement these functions if your add-on doesn't support BFileInterface nodes.

Return **`B_OK`** if you've successfully identified the node; otherwise, return an appropriate error code.

WantsAutoStart()

```
virtual bool WantsAutoStart(void)
```

Implement this hook function to return **`true`** if there are one or more node flavors supported by your add-on that want to be instantiated automatically when the add-on is loaded, instead of waiting until they're explicitly instantiated by an application's request later. Return **`false`** otherwise.

Global C Functions

make_media_addon()

BMediaAddOn ***make_media_addon(**image_id *addonID***)**

`make_media_addon()` is the global C function your add-on needs to implement in order to make itself available to the world. It will be called when the Media Server loads your add-on, and you should return an instance of your BMediaAddOn-derived class, with the appropriate overrides to the hook functions.

The *addonID* argument is your add-on's image ID. Your `make_media_addon()` function should be sure to save the image ID passed to it and pass that value through to the BMediaAddOn constructor, or the Media Kit will be unhappy with you.

This function can be quite simple:

```
BMediaAddOn *make_media_addon(image_id myImage) {
   return new MyConsumerAddOn(myImage);
}
```

It simply creates an instance of the add-on object (in this case, `MyConsumerAddOn`), and returns it.

Structures

dormant_flavor_info

Declared in: be/media/MediaAddOn.h

```
struct dormant_flavor_info : public flavor_info, public BFlattenable {
   dormant_node_info();
   virtual ~dormant_node_info();
   dormant_flavor_info(const dormant_flavor_info &);
   dormant_flavor_info& operator=(const dormant_flavor_info &);
   dormant_flavor_info& operator=(const flavor_info &);

   dormant_node_info node_info;

   void set_name(const char *in_name);
   void set_info(const char *in_info);
   void add_in_format(const media_format &in_format);
   void add_out_format(const media_format &out_format);

   virtual bool IsFixedSize() const;
   virtual type_code TypeCode() const;
   virtual ssize_t FlattenedSize() const;
   virtual status_t Flatten(void *buffer, ssize_t size) const;
   virtual status_t Unflatten(type_code c, const void *buf, ssize_t size);

private:
   void assign_atoms(const flavor_info &that);
   media_addon_id addon;
   int32 flavor_id;
   char name[B_MEDIA_NAME_LENGTH];
private:
   char reserved[128];
};
```

The dormant_flavor_info structure describes a flavor in a dormant node.

dormant_node_info

Declared in: be/media/MediaAddOn.h

```
struct dormant_node_info {
   dormant_node_info();
   ~dormant_node_info();
```

```
   media_addon_id addon;
   int32 flavor_id;
   char name[B_MEDIA_NAME_LENGTH];
private:
   char reserved[128];
};
```

The dormant_node_info structure describes a node that resides in an add-on and that may not be in memory. You might use it when issuing the **BMediaRoster::InstantiateDormantNode()** to instantiate a node, for example.

The *addon* field indicates the ID number of the media add-on in which the node resides, and the *flavor_id* is the flavor ID internal to that add-on indicating which flavor the node should represent. The *name* field is the node's name.

---

flavor_info

Declared in: be/media/MediaAddOn.h

```
struct flavor_info {
   char *name;
   char *info;
   uint64 kinds;
   uint32 flavor_flags;
   int32 internal_id;
   int32 possible_count;

   int32 in_format_count;
   uint32 in_format_flags;
   const media_format *in_formats;

   int32 out_format_count;
   uint32 out_format_flags;
   const media_format *out_formats;

   uint32 _reserved_[16];

private:
   flavor_info & operator=(const flavor_info &other);
};
```

The *name* and *info* fields provide a human-readable name, and information about the flavor.

The *kinds* field should be set to the node kind. See "node_kind" on page202.

The *flavor_flags* field contain flags providing additional information about the flavor.

Constant	Description
`B_FLAVOR_IS_GLOBAL`	The flavor will be forced into the Media Add-on Server, and only one instance of it will exist.
`B_FLAVOR_IS_LOCAL`	The flavor will be forced into the loading application, and many instances of it may exist.

If neither flag is specified, the Media Kit will decide what to do with the flavor.

The *internal_id* is an internal ID number that your add-on can use to identify the flavor; the flavor will be requested by the Media Kit using this ID number.

The *possible_count* field specifies to the Media Kit the maximum number of instances of your node can be in existence at the same time. For example, if your node provides support for a particular sound card, this value should be equal to the number of cards you support that are currently installed in the computer.

in_format_count specifies how many input formats the flavor supports, and *in_formats* is a list of all the input formats supported by the flavor.

in_format_flags provides informational flags about the flavor's inputs. There currently aren't any defined flags, so set this field to 0.

out_format_count specifies how many output formats the flavor supports, and *out_formats* is a list of all the output formats supported by the flavor.

out_format_flags provides informational flags about the flavor's outputs. There currently aren't any defined flags, so set this field to 0.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Media Kit Table of Contents

- The Media Kit
- Reading and Writing Media Files
- A BMediaEventLooper Example
- BBuffer
- BBufferConsumer
- BBufferGroup
- BBufferProducer
- BContinuousParameter
- BControllable
- BDiscreteParameter
- BFileInterface
- BMediaAddOn
- BMediaBufferDecoder
- BMediaBufferEncoder
- BMediaDecoder
- BMediaEncoder
- BMediaEventLooper
- BMediaFile
- BMediaFiles
- BMediaFormats
- BMediaNode
- BMediaRoster
- BMediaTheme
- BMediaTrack
- BNullParameter
- BParameter

- BParameterGroup
- Real-Time Allocation Functions
- BSound
- BTimeCode
- BParameterWeb
- BSmallBuffer
- BSoundPlayer
- BTimedEventQueue
- BTimeSource
- Global Functions
- Message Formats
- Constants and Defined Types

The Be Book **The Media Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Media Kit: Master Index

&

<=	BTimeCode

+

+	BTimeCode
+=	BTimeCode

-

-	BTimeCode
-=	BTimeCode

=

=	BTimeCode
==	BTimeCode

A

AcceptFormat()	A BMediaEventLooper Example
AcceptFormat()	BBufferConsumer

Acquire()	BMediaNode
AddBuffer()	BBufferGroup
AddBuffersTo()	BBufferGroup
AddChunk()	BMediaFile
AddCopyright()	BMediaFile
AddCopyright()	BMediaTrack
AddEvent()	BTimedEventQueue
AddInput()	BParameter
AddItem()	BDiscreteParameter
AddNodeKind()	BMediaNode
AddOn()	A BMediaEventLooper Example
AddOn()	BMediaNode
AddOutput()	BParameter
AddTimer()	BMediaNode
AddTrackInfo()	BMediaEncoder
AddTrackInfo()	BMediaTrack
AdditionalBufferRequested()	BBufferProducer
Advanced Playback	BSoundPlayer
ApplyParameterData()	BControllable
AudioBufferSizeFor()	BMediaRoster
AudioHeader()	BBuffer
AutoStart()	BMediaAddOn
avi_info	Constants and Defined Types

B

B_BALANCE	BParameterGroup
B_BITRATE	BParameterGroup
A BMediaEventLooper Example	A BMediaEventLooper Example
A BMediaEventLooper Example	A BMediaEventLooper Example
A BMediaEventLooper Example	anon
BackgroundBitmapFor()	BMediaTheme
Background Color and Bitmap Kinds	BMediaTheme
BackgroundColorFor()	BMediaTheme
BroadcastChangedParameter()	BControllable
BroadcastNewParameterValue()	BControllable
BroadcastTimeWarp()	BTimeSource
BBuffer	BBuffer
BBuffer	BBuffer
BBufferConsumer	BBufferConsumer
BBufferConsumer()	BBufferConsumer
~BBufferConsumer	BBufferConsumer
BBufferConsumer Functions	A BMediaEventLooper Example
BufferDuration()	BMediaEventLooper
BBufferGroup	BBufferGroup
BBufferGroup()	BBufferGroup
~BBufferGroup	BBufferGroup
BufferPlayer()	BSoundPlayer
BBufferProducer	BBufferProducer

BBufferProducer()	BBufferProducer
BufferReceived()	A BMediaEventLooper Example
BufferReceived()	BBufferConsumer
buffer_clone_info	BBuffer

C

B_COMPRESSION	BParameterGroup
B_CROSSFADE	BParameterGroup
ChangeFormat()	BBufferProducer
CleanUpEvent()	BMediaEventLooper
Cleaning Up After Nodes	BTimedEventQueue
cleanup_flag	BTimedEventQueue
cleanup_hook	BTimedEventQueue
ClipDataToRegion()	BBufferProducer
Clipping Data Formats	BBufferProducer
CloneInfo()	BBuffer
CloseFile()	BMediaFile
Codec Type Info	Constants and Defined Types
B_CODEC_TYPE_INFO	Constants and Defined Types
B_COLOR_SPACE	BParameterGroup
CommitHeader()	BMediaFile
Connect()	A BMediaEventLooper Example
Connect()	BBufferProducer
Connect()	BMediaRoster

connect_flags	BMediaRoster
Connected()	A BMediaEventLooper Example
Connected()	BBufferConsumer
Constants and Defined Types	Constants and Defined Types
Constants and Defined Types	Constants and Defined Types
Constants	BBufferProducer
Constants	BBuffer
Constants	BContinuousParameter
Constants	BMediaEventLooper
Constants	BMediaFile
Constants	BMediaFormats
Constants	BMediaNode
Constants	BMediaRoster
Constants	BMediaTheme
Constants	BMediaTrack
Constants	BParameter
Constants	BSoundPlayer
Constants	BTimeCode
Constants	BTimeSource
Constants	BTimedEventQueue
Constants	Constants and Defined Types
Constructor	A BMediaEventLooper Example
The Constructor and Destructor	A BMediaEventLooper Example
Constructor and Destructor	BBufferConsumer
Constructor and Destructor	BBufferGroup

Constructor and Destructor	BBuffer
Constructor and Destructor	BContinuousParameter
Constructor and Destructor	BDiscreteParameter
Constructor and Destructor	BMediaAddOn
Constructor and Destructor	BMediaBufferDecoder
Constructor and Destructor	BMediaBufferEncoder
Constructor and Destructor	BMediaDecoder
Constructor and Destructor	BMediaEncoder
Constructor and Destructor	BMediaEventLooper
Constructor and Destructor	BMediaFile
Constructor and Destructor	BMediaFiles
Constructor and Destructor	BMediaFormats
Constructor and Destructor	BMediaNode
Constructor and Destructor	BMediaRoster
Constructor and Destructor	BMediaTheme
Constructor and Destructor	BParameterGroup
Constructor and Destructor	BParameterWeb
Constructor and Destructor	BParameter
Constructor and Destructor	BSoundPlayer
Constructor and Destructor	BTimeCode
Constructor and Destructor	BTimedEventQueue
Constructor	BBufferProducer
Constructor	BControllable
Constructor	BFileInterface
Constructor	BSmallBuffer
Constructor	BTimeSource

Constructor Flags	BMediaFile
ConsumerType()	BBufferConsumer
BContinuousParameter	BContinuousParameter
BContinuousParameter	BContinuousParameter
ControlLoop()	BMediaEventLooper
ControlPort()	BMediaNode
ControlThread()	BMediaEventLooper
B_CONTROL_BG	BMediaTheme
B_CONTROL_FG	BMediaTheme
BControllable	BControllable
BControllable()	BControllable
BControllable Functions	A BMediaEventLooper Example
Converting the Files	Reading and Writing Media Files
Cookie()	BSoundPlayer
Copyright()	BMediaFile
CountBuffers()	BBufferGroup
CountChannels()	BParameter
CountFlavors()	BMediaAddOn
CountFrames()	BMediaTrack
CountGroups()	BParameterGroup
CountGroups()	BParameterWeb
CountInputs()	BParameter
CountItems()	BDiscreteParameter
CountOutputs()	BParameter
CountParameters()	BParameterGroup

CountParameters()	BParameterWeb
CountTracks()	BMediaFile
count_timecodes()	BTimeCode
CreateTrack()	BMediaFile
Creating a Parameter Web	A BMediaEventLooper Example
Creating Your Own Node	BMediaNode
CurrentFrame()	BMediaTrack
CurrentRoster()	BMediaRoster
CurrentTime()	BMediaTrack
CurrentTime()	BSoundPlayer

D

Data()	BBuffer
Data Endianness	Constants and Defined Types
Dealing With Multiple Time Sources	BBufferConsumer
Decode()	BMediaDecoder
DecodeBuffer()	BMediaBufferDecoder
DecodedFormat()	BMediaTrack
B_DECREASE_PRECISION	BMediaNode
The Default Pool	Real-Time Allocation Functions
Defined Types	BMediaFormats
Defined Types	BSoundPlayer
Defined Types	BTimeSource

Defined Types	BTimedEventQueue
Defined Types	Constants and Defined Types
DeleteHook()	BMediaNode
Destructor	A BMediaEventLooper Example
Destructor	BMediaTrack
Detecting When Playback Is Complete	BMediaRoster
Disconnect()	BBufferProducer
Disconnect()	BMediaRoster
Disconnected()	A BMediaEventLooper Example
Disconnected()	BBufferConsumer
BDiscreteParameter	BDiscreteParameter
BDiscreteParameter	BDiscreteParameter
DispatchEvent()	BMediaEventLooper
DisposeFileFormatCookie()	BFileInterface
DisposeInputCookie()	A BMediaEventLooper Example
DisposeInputCookie()	BBufferConsumer
DisposeOutputCookie()	BBufferProducer
DoForEach()	BTimedEventQueue
does_file_accept_format()	BMediaFormats
dormant_flavor_info	BMediaAddOn
dormant_node_info	BMediaAddOn
B_DROP_DATA	BMediaNode

Duration()	BMediaTrack

E

B_ENABLE	BParameterGroup
B_EDIT_BG	BMediaTheme
B_EDIT_FG	BMediaTheme
EnableOutput()	BBufferProducer
Encode()	BMediaEncoder
EncodeToBuffer()	BMediaBufferEncoder
encode_parameters	Constants and Defined Types
EncodedFormat()	BMediaTrack
EventCount()	BTimedEventQueue
EventLatency()	BMediaEventLooper
EventNotifier()	BSoundPlayer
EventQueue()	BMediaEventLooper
event_type	BTimedEventQueue
Example: Let's Play Around	BSoundPlayer

F

B_FREQUENCY	BParameterGroup
Families	BMediaFormats
BFileInterface	BFileInterface
BFileInterface()	BFileInterface
FindFirstMatch()	BTimedEventQueue

FindKeyFrameForFrame()	BMediaTrack
FindKeyFrameForTime()	BMediaTrack
FindLatencyFor()	BBufferProducer
FindSeekTag()	BBufferProducer
FirstEvent()	BTimedEventQueue
FirstEventTime()	BTimedEventQueue
Flags	BBuffer
Flags()	BBuffer
flavor_info	BMediaAddOn
Flush()	BMediaTrack
FlushEvents()	BTimedEventQueue
for_each_hook	BTimedEventQueue
ForegroundColorFor()	BMediaTheme
Foreground Color Kinds	BMediaTheme
Format	BMediaFormats
Format()	BSoundPlayer
FormatChangeRequested()	BBufferProducer
FormatChanged()	A BMediaEventLooper Example
FormatChanged()	BBufferConsumer
FormatProposal()	BBufferProducer
FormatSuggestionRequested()	BBufferProducer
format_is_compatible()	Global Functions
B_FRAME_RATE	BParameterGroup
Frames()	BTimeCode
frames_to_timecode()	BTimeCode

G

B_GAIN	BParameterGroup
B_GENERIC	BParameterGroup
B_GENERAL_BG	BMediaTheme
B_GENERAL_FG	BMediaTheme
GetAVIFormatFor()	BMediaFormats
GetAllInputsFor()	BMediaRoster
GetAllOutputsFor()	BMediaRoster
GetAudioInput()	BMediaRoster
GetAudioMixer()	BMediaRoster
GetAudioOutput()	BMediaRoster
GetBeOSFormatFor()	BMediaFormats
GetBufferList()	BBufferGroup
GetCodeFor()	BMediaFormats
GetCodecInfo()	BMediaTrack
GetConfigurationFor()	BMediaAddOn
GetConnectedInputsFor()	BMediaRoster
GetConnectedOutputsFor()	BMediaRoster
GetData()	BTimeCode
GetDecoderInfo()	BMediaDecoder
GetDormantFlavorInfoFor()	BMediaRoster
GetDormantNodes()	BMediaRoster
GetDuration()	BFileInterface
GetEncodeParameters()	BMediaEncoder
GetEncodeParameters()	BMediaTrack

GetFileFormatInfo()	BMediaFile
GetFileFormatList()	BMediaAddOn
GetFileFormatsFor()	BMediaRoster
GetFlavorAt()	BMediaAddOn
GetFormatFor()	BMediaFormats
GetFormatFor()	BMediaRoster
GetFreeInputsFor()	BMediaRoster
GetFreeOutputsFor()	BMediaRoster
GetInitialLatencyFor()	BMediaRoster
GetInstancesFor()	BMediaRoster
GetLatency()	BBufferProducer
GetLatencyFor()	A BMediaEventLooper Example
GetLatencyFor()	BBufferConsumer
GetLatencyFor()	BMediaRoster
GetLiveNodes()	BMediaRoster
GetNextChunk()	BMediaBufferDecoder
GetNextChunk()	BMediaDecoder
GetNextFileFormat()	BFileInterface
GetNextFormat()	BMediaFormats
GetNextInput()	A BMediaEventLooper Example
GetNextInput()	BBufferConsumer
GetNextOutput()	BBufferProducer
GetNextRef()	BMediaFiles
GetNextType()	BMediaFiles
GetNodeAttributes()	BMediaNode
GetNodeAttributesFor()	BMediaRoster

GetNodeFor()	BMediaRoster
GetParameterValue()	A BMediaEventLooper Example
GetParameterValue()	BControllable
GetParameterValue()	BMediaFile
GetParameterValue()	BMediaTrack
GetParameterView()	BMediaFile
GetParameterView()	BMediaTrack
GetParameterWebFor()	BMediaRoster
GetQuality()	BMediaTrack
GetQuicktimeFormatFor()	BMediaFormats
GetReadFileFormatListFor()	BMediaRoster
GetRealtimeFlags()	BMediaRoster
GetRef()	BFileInterface
GetRef()	BMediaTheme
GetRefFor()	BMediaFiles
GetRefFor()	BMediaRoster
GetResponse()	BContinuousParameter
GetStartLatency()	BTimeSource
GetStartLatencyFor()	BMediaRoster
GetString()	BTimeCode
GetSystemTimeSource()	BMediaRoster
GetTime()	BTimeSource
GetTimeSource()	BMediaRoster
GetValue()	BParameter
GetVideoInput()	BMediaRoster
GetVideoOutput()	BMediaRoster

GetVolumeInfo()	BSoundPlayer
GetWriteFileFormatListFor()	BMediaRoster
get_next_encoder()	BMediaFormats
get_next_file_format()	Global Functions
get_nth_theme()	BMediaTheme
get_timecode_description()	BTimeCode
Getting to Node You...	BMediaAddOn
Global C Functions	BMediaAddOn
Global C Functions	BMediaFormats
Global C Functions	BMediaTheme
Global C Functions	BTimeCode
Global C Functions	Real-Time Allocation Functions
Global Functions	Global Functions
Global Functions	Global Functions
Global Functions	Global Functions
B_GOP_SIZE	BParameterGroup
GroupAt()	BParameterGroup
GroupAt()	BParameterWeb
Group()	BParameter

H

HandleBadMessage()	BMediaNode
HandleEvent()	A BMediaEventLooper Example
HandleEvent()	BMediaEventLooper
HandleMessage()	A BMediaEventLooper Example

HandleMessage()	BBufferConsumer
HandleMessage()	BBufferProducer
HandleMessage()	BControllable
HandleMessage()	BFileInterface
HandleMessage()	BMediaNode
HandleMessage()	BTimeSource
HasData()	BSoundPlayer
HasEvents()	BTimedEventQueue
Header()	BBuffer
B_HILITE_BG	BMediaTheme
B_HILITE_FG	BMediaTheme
Hook Function	BParameter
Hook Function	BSoundPlayer
Hook Functions	BBufferConsumer
Hook Functions	BBufferProducer
Hook Functions	BControllable
Hook Functions	BFileInterface
Hook Functions	BMediaAddOn
Hook Functions	BMediaBufferDecoder
Hook Functions	BMediaBufferEncoder
Hook Functions	BMediaDecoder
Hook Functions	BMediaEncoder
Hook Functions	BMediaEventLooper
Hook Functions	BMediaFiles
Hook Functions	BMediaNode
Hook Functions	BMediaTheme

Hook Functions	BTimeSource
Hours()	BTimeCode

I

ID()	BBuffer
ID()	BMediaNode
ID()	BMediaTheme
ID()	BParameter
Identifying an Entry	BMediaFiles
ImageID()	BMediaAddOn
Implementing a Node (the Easy Way)	BMediaEventLooper
B_INCREASE_LATENCY	BMediaNode
Info()	BMediaTheme
InitCheck()	BBufferGroup
InitCheck()	BMediaAddOn
InitCheck()	BMediaBufferDecoder
InitCheck()	BMediaBufferEncoder
InitCheck()	BMediaDecoder
InitCheck()	BMediaEncoder
InitCheck()	BMediaFile
InitCheck()	BMediaFormats
InitCheck()	BMediaTrack
InitCheck()	BSoundPlayer
InputAt()	BParameter

B_INPUT_MUX	BParameterGroup
Installing a Theme Add-on	BMediaTheme
InstantiateDormantNode()	BMediaRoster
InstantiateNodeFor()	BMediaAddOn
Integrating Into a Real Application	Reading and Writing Media Files
IsPlaying()	BSoundPlayer
IsRunning()	BTimeSource
ItemNameAt()	BDiscreteParameter
ItemValueAt()	BDiscreteParameter

K

Keeping Time	BTimeSource
Kind()	BParameter
Kinds()	BMediaNode

L

B_LEVEL	BParameterGroup
LastEvent()	BTimedEventQueue
LastEventTime()	BTimedEventQueue
LateNoticeReceived()	BBufferProducer
Latency()	BSoundPlayer
LatencyChanged()	BBufferProducer
launch_media_server()	Global Functions

LinearFrames()	BTimeCode
live_node_info	Constants and Defined Types
Lock()	BMediaFormats
LockParameterWeb()	BControllable

M

B_MUTE	BParameterGroup
MakeContinuousParameter()	BParameterGroup
MakeControlFor()	BMediaTheme
MakeDiscreteParameter()	BParameterGroup
MakeEmpty()	BDiscreteParameter
MakeFallbackViewFor()	BMediaTheme
MakeFormatFor()	BMediaFormats
MakeGroup()	BParameterWeb
MakeItemsFromInputs()	BDiscreteParameter
MakeItemsFromOutputs()	BDiscreteParameter
MakeNullParameter()	BParameterGroup
MakeParameterData()	BControllable
MakeTimeSourceFor()	BMediaRoster
MakeViewFor()	BMediaTheme
make_format_flags	BMediaFormats
make_media_addon()	BMediaAddOn
make_theme()	BMediaTheme
MALLOC_DEBUG	Real-Time Allocation Functions

B_MASTER_GAIN	BParameterGroup
MaxValue()	BContinuousParameter
BMediaAddOn	BMediaAddOn
BMediaAddOn()	BMediaAddOn
~BMediaAddOn	BMediaAddOn
Media	BMediaFormats
BMediaBufferDecoder	BMediaBufferDecoder
BMediaBufferDecoder()	BMediaBufferDecoder
~BMediaBufferDecoder()	BMediaBufferDecoder
BMediaBufferEncoder	BMediaBufferEncoder
BMediaBufferEncoder()	BMediaBufferEncoder
~BMediaBufferEncoder()	BMediaBufferEncoder
BMediaDecoder	BMediaDecoder
BMediaDecoder()	BMediaDecoder
~BMediaDecoder()	BMediaDecoder
BMediaEncoder	BMediaEncoder
BMediaEncoder()	BMediaEncoder
~BMediaEncoder()	BMediaEncoder
BMediaEventLooper	BMediaEventLooper
BMediaEventLooper()	BMediaEventLooper
~BMediaEventLooper()	BMediaEventLooper
BMediaEventLooper Functions	A BMediaEventLooper Example
BMediaFile	BMediaFile
BMediaFile()	BMediaFile
~BMediaFile()	BMediaFile

BMediaFiles	BMediaFiles
BMediaFiles()	BMediaFiles
~BMediaFiles()	BMediaFiles
MediaFlags()	BMediaRoster
BMediaFormats	BMediaFormats
BMediaFormats()	BMediaFormats
~BMediaFormats()	BMediaFormats
Media Message Size	Constants and Defined Types
BMediaNode	BMediaNode
BMediaNode()	BMediaNode
~BMediaNode	BMediaNode
BMediaNode Functions	A BMediaEventLooper Example
BMediaRoster	BMediaRoster
BMediaRoster()	BMediaRoster
~BMediaRoster	BMediaRoster
BMediaTheme	BMediaTheme
BMediaTheme()	BMediaTheme
~BMediaTheme()	BMediaTheme
BMediaTrack	BMediaTrack
BMediaTrack	BMediaTrack
~BMediaTrack()	BMediaTrack
MediaType()	BParameter
media_addon_id	Constants and Defined Types

media_aiff_description	BMediaFormats
B_MEDIA_ANY_QUALITY	Constants and Defined Types
media_asf_description	BMediaFormats
media_audio_header	Constants and Defined Types
media_avi_description	BMediaFormats
media_avr_description	BMediaFormats
media_beos_description	BMediaFormats
B_MEDIA_BIG_ENDIAN	Constants and Defined Types
B_MEDIA_BUFFER_CREATED	Message Formats
B_MEDIA_BUFFER_DELETED	Message Formats
media_buffer_id	Constants and Defined Types
media_codec_info	BMediaFormats
B_MEDIA_CONNECTION_BROKEN	Message Formats
B_MEDIA_CONNECTION_MADE	Message Formats
media_decode_info	Constants and Defined Types
media_destination	Constants and Defined Types
media_display_flags	Constants and Defined Types
media_encode_info	Constants and Defined Types
media_encoded_audio_format	Constants and Defined Types

media_encoded_audio_header	Constants and Defined Types
media_encoded_video_format	Constants and Defined Types
media_encoded_video_header	Constants and Defined Types
media_file_format	Constants and Defined Types
media_file_format_id	Constants and Defined Types
media_file_format_info	BMediaFormats
media_flags	Constants and Defined Types
B_MEDIA_FLAVORS_CHANGED	Message Formats
media_format	Constants and Defined Types
B_MEDIA_FORMAT_CHANGED	Message Formats
media_format_description	BMediaFormats
media_format_family	Constants and Defined Types
media_format_flags	Constants and Defined Types
media_header	Constants and Defined Types
B_MEDIA_HIGH_QUALITY	Constants and Defined Types
media_input	Constants and Defined Types

B_MEDIA_LITTLE_ENDIAN	Constants and Defined Types
B_MEDIA_LOW_QUALITY	Constants and Defined Types
B_MEDIA_MEDIUM_QUALITY	Constants and Defined Types
media_misc_description	BMediaFormats
media_mpeg_description	BMediaFormats
media_multi_audio_format	Constants and Defined Types
media_multi_audio_info	Constants and Defined Types
media_multi_matrix	Constants and Defined Types
media_multistream_format	Constants and Defined Types
media_multistream_header	Constants and Defined Types
B_MEDIA_NEW_PARAMETER_VALUE	Message Formats
media_node	Constants and Defined Types
media_node_attribute	Constants and Defined Types
B_MEDIA_NODE_CREATED	Message Formats
B_MEDIA_NODE_DELETED	Message Formats
media_node_id	Constants and Defined Types
media_output	Constants and Defined

	Types
B_MEDIA_PARAMETER_CHANGED	Message Formats
media_parameter_type	BParameter
media_producer_status	Constants and Defined Types
media_quicktime_description	BMediaFormats
media_raw_audio_format	Constants and Defined Types
media_raw_video_format	Constants and Defined Types
media_realtime_flags	Constants and Defined Types
media_realtime_init_image()	Global Functions
media_realtime_init_thread()	Global Functions
media_request_info	Constants and Defined Types
media_seek_tag	Constants and Defined Types
media_seek_type	BMediaTrack
media_source	Constants and Defined Types
media_timed_event	BTimedEventQueue
media_type	Constants and Defined Types
media_video_display_info	Constants and Defined Types
media_video_header	Constants and Defined Types

media_wav_description	BMediaFormats
B_MEDIA_WEB_CHANGED	Message Formats
Member Functions	BBufferConsumer
Member Functions	BBufferGroup
Member Functions	BBufferProducer
Member Functions	BBuffer
Member Functions	BContinuousParameter
Member Functions	BControllable
Member Functions	BDiscreteParameter
Member Functions	BFileInterface
Member Functions	BMediaAddOn
Member Functions	BMediaBufferDecoder
Member Functions	BMediaBufferEncoder
Member Functions	BMediaDecoder
Member Functions	BMediaEncoder
Member Functions	BMediaEventLooper
Member Functions	BMediaFile
Member Functions	BMediaFiles
Member Functions	BMediaFormats
Member Functions	BMediaNode
Member Functions	BMediaRoster
Member Functions	BMediaTheme
Member Functions	BMediaTrack
Member Functions	BNullParameter
Member Functions	BParameterGroup
Member Functions	BParameterWeb

Member Functions	BParameter
Member Functions	BSoundPlayer
Member Functions	BTimeCode
Member Functions	BTimeSource
Member Functions	BTimedEventQueue
Memory Pools	Real-Time Allocation Functions
Message Formats	Message Formats
Message Formats	Message Formats
Microseconds()	BTimeCode
MinValue()	BContinuousParameter
Minutes()	BTimeCode
multi_media_channels	Constants and Defined Types

N

Name()	BMediaNode
Name()	BMediaTheme
Name()	BParameterGroup
Name()	BParameter
Negotiating a Connection	BMediaNode
NewChangeTag()	BMediaNode
Node()	BMediaNode
Node()	BParameterWeb
NodeIDFor()	BMediaRoster

NodeRegistered()	A BMediaEventLooper Example
NodeRegistered()	BMediaEventLooper
NodeRegistered()	BMediaNode
NodeStopped()	BMediaNode
node_error	BMediaNode
node_kind	BMediaNode
node_kind	BMediaNode
node_kind	Constants and Defined Types
Notification Message "what" Codes	Message Formats
Notification Messages	Message Formats
NotifyLateProducer()	BBufferConsumer
Now()	BTimeSource
BNullParameter	BNullParameter
BNullParameter	BNullParameter

O

B_OFFLINE	BMediaNode
OfflineTime()	BMediaEventLooper
Operators	BTimeCode
Other BMediaNode Functions	A BMediaEventLooper Example
OutputAt()	BParameter
B_OUTPUT_MUX	BParameterGroup

P

ParameterAt()	BParameterGroup
ParameterAt()	BParameterWeb
BParameter	BParameter
BParameter	BParameter
BParameterGroup	BParameterGroup
BParameterGroup()	BParameterGroup
~BParameterGroup()	BParameterGroup
Parameter Kinds	BParameter
BParameterWeb	BParameterWeb
BParameterWeb()	BParameterWeb
~BParameterWeb()	BParameterWeb
PerformanceTime()	BSoundPlayer
PerformanceTimeFor()	BTimeSource
play_id	BSoundPlayer
play_sound()	Global Functions
Playing Media from Disk	BMediaRoster
Playing Sound	BSoundPlayer
PreferredTheme()	BMediaTheme
PrepareToConnect()	BBufferProducer
Preparing a media_format	Reading and Writing Media Files
Preroll()	BMediaNode
PrerollNode()	BMediaRoster
B_PRESENTATION_BG	BMediaTheme
B_PRESENTATION_FG	BMediaTheme

Priority()	BMediaEventLooper
ProducerDataStatus()	A BMediaEventLooper Example
ProducerDataStatus()	BBufferConsumer
ProducerType()	BBufferProducer
Producer-specific Issues	A BMediaEventLooper Example
ProposeFormatChange()	BBufferProducer
PublishTime()	BTimeSource

Q

B_QUALITY	BParameterGroup
Quality	Constants and Defined Types
queue_action	BTimedEventQueue
Quit()	BMediaEventLooper

R

B_RECORDING	BMediaNode
B_RECSTATE	BParameterGroup
B_RESOLUTION	BParameterGroup
ReadChunk()	BMediaTrack
ReadFrames()	BMediaTrack
Reading and Writing Media Files	anon
Reading and Writing Media Files	Reading and Writing Media Files
Reading and Writing Media Files	Reading and Writing Media Files

RealTime()	BTimeSource
RealTimeFor()	BTimeSource
RealTimeQueue()	BMediaEventLooper
Real-Time Allocation Functions	Real-Time Allocation Functions
Real-Time Allocation Functions	Real-Time Allocation Functions
Realtime Allocators and Thread Locking	BMediaNode
ReclaimAllBuffers()	BBufferGroup
Recycle()	BBuffer
RegionToClipData()	BBufferConsumer
RegisterNode()	BMediaRoster
ReleaseAllTracks()	BMediaFile
Release()	BMediaNode
ReleaseNode()	BMediaRoster
ReleaseTrack()	BMediaFile
RemoveEvent()	BTimedEventQueue
RemoveFirstEvent()	BTimedEventQueue
RemoveItem()	BMediaFiles
RemoveRefFor()	BMediaFiles
ReplaceFrames()	BMediaTrack
ReportError()	BMediaNode
RequestAdditionalBuffer()	BBufferConsumer
RequestBuffer()	BBufferGroup
RequestCompleted()	BMediaNode

RequestError()	BBufferGroup
RequestFormatChange	BBufferConsumer
response	BContinuousParameter
The Rest of the Story...	A BMediaEventLooper Example
RewindFormats()	BMediaFormats
RewindRefs()	BMediaFiles
RewindTypes()	BMediaFiles
RollNode()	BMediaRoster
Roster()	BMediaRoster
rtm_alloc()	Real-Time Allocation Functions
rtm_create_pool()	Real-Time Allocation Functions
rtm_default_pool()	Real-Time Allocation Functions
rtm_delete_pool()	Real-Time Allocation Functions
rtm_free()	Real-Time Allocation Functions
rtm_phys_size_for()	Real-Time Allocation Functions
rtm_realloc()	Real-Time Allocation Functions
rtm_size_for()	Real-Time Allocation Functions
Run()	BMediaEventLooper
RunMode()	BMediaNode

RunState()	BMediaEventLooper
run_mode	BMediaNode
run_mode	BMediaNode
run_state	BMediaEventLooper

S

SchedulingLatency()	BMediaEventLooper
Seconds()	BTimeCode
Seek()	BMediaNode
Seek()	BTimeSource
SeekNode()	BMediaRoster
SeekTagRequested()	A BMediaEventLooper Example
SeekTagRequested()	BBufferConsumer
Seek Tags	BBufferProducer
SeekTimeSource()	BMediaRoster
SeekToFrame()	BMediaTrack
SeekToTime()	BMediaTrack
SendBuffer()	BBufferProducer
SendDataStatus()	BBufferProducer
SendLatencyChange()	BBufferConsumer
SendRunMode()	BTimeSource
SetAudioInput()	BMediaRoster
SetAudioOutput()	BMediaRoster
SetBufferDuration()	BMediaEventLooper

SetBufferGroup()	BBufferProducer
SetBufferPlayer()	BSoundPlayer
SetCallbacks()	BSoundPlayer
SetChannelCount()	BParameter
SetCleanupHook()	BTimedEventQueue
SetCookie()	BSoundPlayer
SetData()	BTimeCode
SetEncodeParameters()	BMediaEncoder
SetEncodeParameters()	BMediaTrack
SetEventLatency()	BMediaEventLooper
SetEventNotifier()	BSoundPlayer
SetFormat()	BMediaEncoder
SetHasData()	BSoundPlayer
SetInitError()	BSoundPlayer
SetInitialLatency()	BBufferProducer
SetInputFormat()	BMediaDecoder
SetLinearFrames()	BTimeCode
SetMediaType()	BParameter
SetMicroseconds()	BTimeCode
SetOfflineTime()	BMediaEventLooper
SetOutputBuffersFor()	BBufferConsumer
SetOutputEnabled()	BBufferConsumer
SetOutputFormat	BMediaDecoder
SetParameterValue()	A BMediaEventLooper Example
SetParameterValue()	BControllable

SetParameterValue()	BMediaFile
SetParameterValue()	BMediaTrack
SetParameterWeb()	BControllable
SetPlayRate()	BBufferProducer
SetPreferredTheme()	BMediaTheme
SetPriority()	BMediaEventLooper
SetProducerRate()	BMediaRoster
SetProducerRunModeDelay()	BMediaRoster
SetQuality()	BMediaTrack
SetRealtimeFlags()	BMediaRoster
SetRef()	BFileInterface
SetRefFor()	BMediaFiles
SetRefFor()	BMediaRoster
SetResponse()	BContinuousParameter
SetRunMode()	BMediaNode
SetRunMode()	BTimeSource
SetRunModeNode()	BMediaRoster
SetRunState()	BMediaEventLooper
SetSizeUsed()	BBuffer
SetSoundVolume()	BSoundPlayer
SetTimeSource()	BMediaNode
SetTimeSourceFor()	BMediaRoster
SetTo()	BMediaDecoder
SetTo()	BMediaEncoder
SetType()	BTimeCode
SetValue()	BParameter

SetVideoClippingFor()	BBufferConsumer
SetVideoInput()	BMediaRoster
SetVideoOutput()	BMediaRoster
SetVolume()	BSoundPlayer
SetVolumeDB()	BSoundPlayer
B_SETTINGS_BG	BMediaTheme
B_SETTINGS_FG	BMediaTheme
shutdown_media_server()	Global Functions
B_SHUTTLE_MODE	BParameterGroup
B_SHUTTLE_SPEED	BParameterGroup
SizeAvailable()	BBuffer
SizeUsed()	BBuffer
BSmallBuffer	BSmallBuffer
BSmallBuffer()	BSmallBuffer
SmallBufferSizeLimit()	BSmallBuffer
SniffRef()	BFileInterface
SniffRef()	BMediaAddOn
SniffRef()	BMediaRoster
SniffRefFor()	BMediaRoster
SniffType()	BMediaAddOn
SniffTypeKind()	BMediaAddOn
SnoozeUntil()	BTimeSource
BSoundPlayer	BSoundPlayer
BSoundPlayer()	BSoundPlayer
~BSoundPlayer()	BSoundPlayer
sound_player_notification	BSoundPlayer

Special Considerations	BMediaNode
Start()	BMediaNode
Start()	BSoundPlayer
Start()	BTimeSource
StartControlPanel()	BControllable
StartControlPanel()	BMediaRoster
StartNode()	BMediaRoster
StartPlaying()	BSoundPlayer
StartTimeSource()	BMediaRoster
StartWatching()	BMediaRoster
Start(), Stop(), Seek(), TimeWarp()	A BMediaEventLooper Example
Static Functions	BSmallBuffer
Stop()	BMediaNode
Stop()	BSoundPlayer
Stop()	BTimeSource
StopNode()	BMediaRoster
StopPlaying()	BSoundPlayer
StopTimeSource()	BMediaRoster
StopWatching()	BMediaRoster
stop_sound()	Global Functions
string_for_format()	Global Functions
Structures	BBuffer
Structures	BMediaAddOn
Structures	BTimeCode
suggestion_quality	BBufferProducer

suggestion_quality	BBufferProducer
Supported Families	BMediaFormats
SyncToNode()	BMediaRoster

T

B_TRACK	BParameterGroup
BTimeCode	BTimeCode
BTimeCode()	BTimeCode
~BTimeCode()	BTimeCode
TimeSource()	BMediaNode
BTimeSource	BTimeSource
BTimeSource()	BTimeSource
TimeSourceOp()	BTimeSource
TimeWarp()	BMediaNode
time_direction	BTimedEventQueue
time_source_op	BTimeSource
time_source_op_info	BTimeSource
B_TIMECODE_100	BTimeCode
B_TIMECODE_18	BTimeCode
B_TIMECODE_24	BTimeCode
B_TIMECODE_25	BTimeCode
B_TIMECODE_30	BTimeCode
B_TIMECODE_30_DROP_2	BTimeCode
B_TIMECODE_30_DROP_4	BTimeCode
B_TIMECODE_75	BTimeCode

B_TIMECODE_DEFAULT	BTimeCode
timecode_info	BTimeCode
timecode_to_frames()	BTimeCode
timecode_to_us()	BTimeCode
timecode_type	BTimeCode
BTimedEventQueue	BTimedEventQueue
BTimedEventQueue()	BTimedEventQueue
~BTimedEventQueue()	BTimedEventQueue
TimerExpired()	BMediaNode
TrackAt()	BMediaFile
B_TUNER_CHANNEL	BParameterGroup
Type()	BBuffer
Type()	BParameter
Type()	BTimeCode

U

Unit()	BParameter
Unlock()	BMediaFormats
UnlockParameterWeb()	BControllable
UnregisterNode()	BMediaRoster
us_to_timecode()	BTimeCode
Using BBitmaps as Buffers	BBufferGroup
Using BMediaRoster Functions from Nodes	BMediaRoster
Using BSoundPlayer	BSoundPlayer

Using the transcode() Function	Reading and Writing Media Files

V

ValueStep()	BContinuousParameter
ValueType()	BContinuousParameter
ValueType()	BDiscreteParameter
ValueType()	BNullParameter
ValueType()	BParameter
vid_info	Constants and Defined Types
Video Clipping	BBufferProducer
Video Clipping	BBufferProducer
VideoClippingChanged()	BBufferProducer
VideoHeader()	BBuffer
video_orientation	Constants and Defined Types
ViewFor()	BMediaTheme
Volume()	BSoundPlayer
VolumeDB()	BSoundPlayer

W

WaitForBuffers()	BBufferGroup
WaitForMessage()	BMediaNode
WaitForSound()	BSoundPlayer
wait_for_sound()	Global Functions

WantsAutoStart()	BMediaAddOn
Web()	BControllable
Web()	BMediaFile
Web()	BMediaTrack
Web()	BParameterGroup
Web()	BParameter
B_WEB_ADC_CONVERTER	BParameterGroup
B_WEB_BUFFER_INPUT	BParameterGroup
B_WEB_BUFFER_OUTPUT	BParameterGroup
B_WEB_DAC_CONVERTER	BParameterGroup
B_WEB_LOGICAL_INPUT	BParameterGroup
B_WEB_LOGICAL_OUTPUT	BParameterGroup
B_WEB_PHYSICAL_INPUT	BParameterGroup
B_WEB_PHYSICAL_OUTPUT	BParameterGroup
WriteChunk()	BMediaBufferEncoder
WriteChunk()	BMediaEncoder
WriteChunk()	BMediaTrack
WriteFrames()	BMediaTrack

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimeCode

Derived from: none
Declared in: be/media/TimeCode.h **Library:** libmedia.so
Summary: *more...*

Include here information about types etc.

Constructor and Destructor

BTimeCode()

BTimeCode()

BTimeCode(bigtime_t *us*,
    timecode_type *type* = B_TIMECODE_DEFAULT**)**
BTimeCode(const BTimeCode &*clone***)**
BTimeCode(int *hours*, int *minutes*, int *seconds*, int *frames*,
    timecode_type *type* = B_TIMECODE_DEFAULT**)**

The constructor prepares the BTimeCode object for use. If you use the first form of the constructor, without arguments, you'll have to call an appropriate function to set the BTimeCode's time information before using it for translation purposes. This can be done by calling one or more of **SetData()**, **SetType()**, **SetMicroseconds()**, or **SetLinearFrames()**.

The second form of the constructor accepts as input a time in microseconds, *us*, and the timecode *type*.

The third form of the constructor duplicates an existing BTimeCode object.

The fourth form accepts as input a time in *hours*, *minutes*, *seconds*, and *frames*, as well as the timecode *type*.

~BTimeCode()

~BTimeCode()

A typical destructor

Member Functions

Frames() see Hours()

GetData() , SetData()

```
void GetData(int *outHours,
    int *outMinutes,
    int *outSeconds,
    int *outFrames,
    timecode_type *outType = NULL)
```

```
void SetData(int hours,
    int minutes,
    int seconds,
    int frames)
```

GetData() returns the timecode's value in hours, minutes, seconds, and frames, and also returns the timecode type, if you specify a valid pointer for *outType*. SetData() lets you set the timecode's value.

GetString()

```
void GetString(char *str) const
```

Fills str, which must be at least 24 bytes long, with a string indicating the current time in hours, minutes, seconds, and frames. The string is formatted in a manner appropriate to the timecode type. A typical example would be "01:24:09.18", which is 1 hour, 24 minutes, 9 seconds, and 18 frames.

Hours() , Minutes() , Seconds() , Frames()

```
int Hours(void) const
```

```
int Minutes(void) const
```

```
int Seconds(void) const
```

```
int Frames(void) const
```

These functions return the time's hours, minutes, seconds, and frames portions.

LinearFrames() , SetLinearFrames()

```
int32 LinearFrames(void) const
void SetLinearFrames(int32 linearFrames)
```

LinearFrames() returns the BTimeCode object's time in linear frames. SetLinearFrames() lets you change the time, specifying the new time in linear frames.

Microseconds() , SetMicroseconds()

```
bigtime_t Microseconds(void) const
void SetMicroseconds(bigtime_t us)
```

Microseconds() returns the BTimeCode object's time, in microseconds. SetMicroseconds() lets you change the time, specifying the new time in microseconds.

- Minutes() see Hours()
- Seconds() see Hours()
- SetData() see GetData()
- Set see LinearFrames()
- Set see Microseconds()
- Set see Type()

Type() , SetType()

```
timecode_type Type(void) const
status_t SetType(timecode_type type)
```

Type() returns the BTimeCode object's timecode type. SetType() lets you change the timecode type.

RETURN CODES

- B_OK. The type was returned safely.
- B_ERROR. The timecode type specified is invalid.

Operators

= (assignment)

```
BTimeCode &operator =(const BTimeCode &clone)
```

Makes the current BTimeCode identical to the BTimeCode object specified

== (equality)

```
BTimeCode &operator ==(const BTimeCode &other)
```

Determines whether or not the two BTimeCode objects are equal (their times are the same, regardless of their timecode types).

< (less than)

```
BTimeCode &operator <(const BTimeCode &other)
```

Indicates whether or not one BTimeCode's time, in microseconds, is less than the other's.

+= (expression plus)

```
BTimeCode &operator +=(const BTimeCode &other)
```

Adds the time of the BTimeCode object *other* to the current BTimeCode's time.

-= (expression minus)

```
BTimeCode &operator -=(const BTimeCode &other)
```

Subtracts the time of the BTimeCode object *other* from the current BTimeCode's time.

+ (plus)

```
BTimeCode &operator +(const BTimeCode &other)
```

Adds two BTimeCode values together, returning a new BTimeCode.

- (minus)

BTimeCode &**operator -**(const BTimeCode &*other*)

Subtracts two BTimeCode values, returning a new one.

Global C Functions

count_timecodes()

```
status_t count_timecodes(void)
```

Returns the number of recognized time code types.

frames_to_timecode() , timecode_to_frames()

```
status_t frames_to_timecode(int32 linearFrames,
    int * hours, int * minutes, int * seconds, int * frames,
    const timecode_info * code = NULL)
```

```
status_t timecode_to_frames(int hours, int minutes, int seconds, int frames,
    int32 * linearFrames,
    const timecode_info * code = NULL)
```

`frames_to_timecode()` converts the frame offset *linearFrames* into *hours*, *minutes*, *seconds*, and *frames*.

`timecode_to_frames()` converts the time from *hours*, *minutes*, *seconds*, and *frames* into a linear frame offset, storing the result in *linearFrames*.

The timecode_info structure *code* is used to determine how the conversion should be made, if you specify it. Otherwise `B_TIMECODE_DEFAULT` is assumed.

Currently these functions always return `B_OK`, but you should still check for errors because you'd hate it if your app broke in the future, wouldn't you?

get_timecode_description()

```
status_t get_timecode_description(timecode_type type,
    timecode_info * outTimeCode)
```

Fills out the timecode_info structure specified by *outTimeCode* with information describing the specified timecode *type*.

RETURN CODES

- `B_OK`. No error.
- `B_ERROR`. The specified timecode type isn't valid.

timecode_to_frames() see frames_to_timecode()

timecode_to_us() see us_to_timecode()

---

us_to_timecode() , timecode_to_us()

```
status_t us_to_timecode(bigtime_t micros,
    int * hours, int * minutes, int * seconds, int * frames,
    const timecode_info * code = NULL)
```

```
status_t timecode_to_us(int hours, int minutes, int seconds, int frames,
    bigtime_t * micros, const timecode_info * code = NULL)
```

`us_to_timecode()` converts the time *micros*, which is specified in microseconds, into *hours*, *minutes*, *seconds*, and *frames*.

`timecode_to_us()` converts the time from *hours*, *minutes*, *seconds*, and *frames* into microseconds, storing the result in *micros*.

The timecode_info structure *code* is used to determine how the conversion should be made, if you specify it. Otherwise `B_TIMECODE_DEFAULT` is assumed.

Currently these functions always return `B_OK`, but you should still check for errors because you'd hate it if your app broke in the future, wouldn't you?

Constants

timecode_type

Declared in: be/media/TimeCode.h

Constant	Description
B_TIMECODE_DEFAULT	The default time code
B_TIMECODE_100	100 frames per second.
B_TIMECODE_75	CD audio
B_TIMECODE_30	MIDI
B_TIMECODE_30_DROP_2	NTSC
B_TIMECODE_30_DROP_4	Brazil
B_TIMECODE_25	PAL
B_TIMECODE_24	Film
B_TIMECODE_18	Super8

Constants identifying the various timecode types supported by BTimeCode.

Structures

timecode_info

Declared in: be/media/TimeCode.h

```
struct timecode_info {
    timecode_type type;
    int drop_frames;
    int every_nth;
    int except_nth;
    int fps_div;
    char name[32];
    char format[32];
    char _reserved_[64];
    };
```

The timecode_info structure describes the attributes of a timecode type. You probably should just use the BTimeCode class, or the global C functions, though. It just makes your life easier.

- *type* indicates the timecode type described by the structure.
- *drop_frames* indicates how many frames this timecode drops every *every_nth* minute, except *every_nth* minute.
- *fps_div* indicates the nominal frame rate of the format.
- *name* is a printable name that can be used in constructing user interfaces.
- *format* is a format to be used in calling **sprintf()**; it's used by **GetString()**.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimeCode Function Summary

Function Summary

Function	Description
<	Determines if one BTimeCode's time is lower than another's
+	Adds another BTimeCode's value
+=	Adds another BTimeCode's value
-	Subtracts another BTimeCode's value
-=	Subtracts another BTimeCode's value
=	Makes one BTimeCode equivalent to another
==	Determines if two BTimeCode objects are equal
count_timecodes()	Returns the number of recognized time code types
Frames()	Returns the frames portion of the time
frames_to_timecode()	Given a frame offset and a timecode_info structure, returns the time in hours, minutes, seconds, and frames
GetData()	Gets the timecode's value in hours, minutes, seconds, and frames

GetString()	Returns the time in string form
get_timecode_description()	Returns a timecode_info structure for the specified timecode type
Hours()	Returns the hours portion of the time
LinearFrames()	Returns the time in terms of linear frames
Microseconds()	Returns the time in microseconds
Minutes()	Returns the minutes portion of the time
Seconds()	Returns the seconds portion of the time
SetData()	Sets the timecode's value in hours, minutes, seconds, and frames
SetLinearFrames()	Sets the time in terms of linear frames
SetMicroseconds()	Sets the time in microseconds
SetType()	Sets the timecode type
BTimeCode()	Basic constructor
~BTimeCode()	Destructor
timecode_to_frames()	Given a time in hours, minutes, seconds, and frames, and a timecode_info structure, returns the linear offset in frames

timecode_to_us()	Given a time in hours, minutes, seconds, and frames, and a timecode_info structure, returns the corresponding time in microseconds
Type()	Returns the timecode type
us_to_timecode()	Given a time in microseconds and a timecode_info structure, returns the corresponding time in hours, minutes, seconds, and frames

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Error Codes

Declared in: be/support/Errors.h
more...

Error codes are returned by various functions to indicate the success or to describe the failure of a requested operation.

All Be error constants except for **B_OK** (**B_NO_ERROR**) are negative integers; any function that returns an error code can thus be lazily tested for success or failure by the following:

```
if ( funcCall() < B_NO_ERROR )
   /* failure*/
else
   /* success*/
```

All constants (except **B_NO_ERROR** and **B_ERROR**) are less than or equal to the value of the **B_ERRORS_END** constant. If you want to define your own negative-valued error codes, you should begin with the value (**B_ERRORS_END** + 1) and work your way toward 0.

POSIX Errors

The BeOS supports the POSIX error code constants (these constants start with the letter "E", as in **EBADF** or **ENOENT**). A number of Be-defined constants are synonyms for the POSIX constants; for example, the Be equivalent for **ENOENT** is **B_ENTRY_NOT_FOUND**.

Most of the General Error Codes, and all the File System Error Codes are covers for POSIX errors. The POSIX equivalents are listed where applicable.

The POSIX constants, and the Be synonyms, can be passed to the POSIX **strerror()** function. The function, defined in **posix/string.h**, returns a human-readable description of the error:

```
char *strerror(int error_code)
```

General Error Codes

Code	POSIX	Description
B_NO_MEMORY	**ENOMEM**	There isn't enough memory for the operation
B_IO_ERROR	**EIO**	A general input/output error occurred
B_PERMISSION_DENIED	**EACCESS**	Illegal access
B_BAD_INDEX		The index is out of range
B_BAD_VALUE	**EINVAL**	An illegal value was passed to the function
B_MISMATCHED_VALUE S		Conflicting values were passed to the function
B_BAD_TYPE		An illegal argument type was named or passed
B_NAME_NOT_FOUND		There's no match for the specified name.
B_NAME_IN_USE		The requested (unique) name is already used
B_BUSY	**EBUSY**	A device is busy, or a file is locked
B_NOT_ALLOWED	**EPERM**	Operation not allowed
B_NO_INIT		An object or structure isn't properly initialized
B_TIMED_OUT	**ETIMEDOUT**	Time expired before the operation was finished
B_INTERRUPTED	**EINTR**	A signal interrupted the operation
B_WOULD_BLOCK	**EAGAIN**	But you don't want to block
B_WOULD_BLOCK	**EWOULD BLOCK**	Same as the above
B_ERROR = -1		A convenient catchall for general errors
B_NO_ERROR = 0		Everything's OK
B_OK		Same as **B_NO_ERROR**

File System Error Codes

Code	POSIX	Description
B_FILE_ERROR	EBADF	A file error occurred
B_FILE_EXISTS	EEXIST	And you don't want to clobber it
B_ENTRY_NOT_FOUND	ENOENT	The requested entry doesn't exist
B_NAME_TOO_LONG	ENAMETOOLONG	Leaf or pathname too long
B_NO_MORE_FDS	EMFILE	No more file descriptors
B_NOT_A_DIRECTORY	ENOTDIR	When it should have been
B_IS_A_DIRECTORY	EISDIR	When it shouldn't be
B_DIRECTORY_NOT_EMPTY	ENOTEMPTY	Attempt to delete a non-empty directory
B_DEVICE_FULL	ENOSPC	Full disk
B_READ_ONLY_DEVICE	EROFS	Write request on a read-only file system
B_CROSS_DEVICE_LINK	EXDEV	Hard link across devices not allowed
B_LINK_LIMIT	ELOOP	Nested links too deep
B_BUSTED_PIPE	EPIPE	Pipe no longer functional.
B_UNSUPPORTED		An unsupported operation was attempted.
B_PARTITION_TOO_SMALL		A partition is too small for the attempted operation.

Application Kit Error Codes

Code	Description
B_DUPLICATE_REPLY	A reply message has already been sent
B_BAD_REPLY	The reply message is inappropriate and can't be sent
B_BAD_HANDLER	The designated message handler isn't valid
B_MESSAGE_TO_SELF	A thread is trying to send a message to itself
B_ALREADY_RUNNING	The application can't be launched again
B_LAUNCH_FAILED	The attempt to launch the application failed
B_AMBIGUOUS_APP_LAUNCH	Odd things happening in app land
B_UNKOWN_MIME_TYPE	Application signature unknown
B_BAD_SCRIPT_SYNTAX	Script syntax malformed
B_LAUNCH_FAILED_NO_RESOLVE_LINK	Unable to resolve a link to the application to be launched
B_LAUNCH_FAILED_EXECUTABLE	The attempt to launch the application failed because the application isn't a valid executable
B_LAUNCH_FAILED_APP_NOT_FOUND	The attempt to launch the application failed because the application wasn't found
B_LAUNCH_FAILED_APP_IN_TRASH	The attempt to launch the application failed because the application was in the trash
B_LAUNCH_FAILED_NO_PREFERRED_APP	The attempt to launch the application failed because there is no preferred application for the document
B_LAUNCH_FAILED_FILES_APP_NOT_FOUND	The attempt to launch the application failed because the document's application is missing
B_BAD_MIME_SNIFFER_RULE	A MIME type sniffer rule is invalid

Kernel Kit Error Codes

Code	Description
B_BAD_THREAD_ID	Specified thread identifier (**thread_id**) is invalid
B_BAD_THREAD_STATE	The thread is in the wrong state for the operation
B_NO_MORE_THREADS	All thread identifiers are currently taken
B_BAD_TEAM_ID	Specified team identifier (**team_id**) is invalid
B_NO_MORE_TEAMS	All team identifiers are currently taken
B_BAD_PORT_ID	Specified port identifier (**port_id**) is invalid
B_NO_MORE_PORTS	All port identifiers have been taken
B_BAD_SEM_ID	Semaphore identifier (**sem_id**) is invalid
B_NO_MORE_SEMS	All semaphores are currently taken
B_BAD_IMAGE_ID	Specified image identifier (**image_id**)is invalid
B_NOT_AN_EXECUTABL E	An executable image was expected
B_BAD_ADDRESS	Illegal address
B_MISSING_LIBRARY	A library needed by an application is missing
B_MISSING_SYMBOL	A symbol needed by an application is missing
B_DEBUGGER_ALREAD Y_INSTALLED	An attempt was made to install the debugger when it was already installed.

Media Kit Error Codes

Code	Description
B_MEDIA_SYSTEM_FAILURE	The media system has failed spectacularly
B_MEDIA_BAD_NODE	An invalid or nonexistent node was specified
B_MEDIA_NODE_BUSY	The specified node was busy
B_MEDIA_BAD_FORMAT	The specified media_format is invalid
B_MEDIA_TOO_MANY_NODE S	The media system has reached its limit of nodes
B_MEDIA_TOO_MANY_BUFF ERS	The media system has too many buffers
B_MEDIA_NODE_ALREADY_E XISTS	Trying to create a node that already exists is a no-no
B_MEDIA_BUFFER_ALREADY _EXISTS	Trying to create a buffer that already exists is a bad idea
B_MEDIA_CANNOT_SEEK	A seek failure occurred
B_MEDIA_CANNOT_CHANGE _RUN_MODE	Unable to change run mode
B_MEDIA_APP_ALREADY_RE GISTERED	An application has already registered with the media system
B_MEDIA_APP_NOT_REGIST ERED	An application hasn't registered with the media system
B_MEDIA_CANNOT_RECLAIM _BUFFERS	Reclaiming of BBuffers failed
B_MEDIA_BUFFERS_NOT_RE CLAIMED	BBuffers weren't reclaimed
B_MEDIA_TIME_SOURCE_ST OPPED	The time source isn't running
B_MEDIA_TIME_SOURCE_BU SY	The time source is busy
B_MEDIA_BAD_SOURCE	The specified media_source is invalid
B_MEDIA_BAD_DESTINATION	The specified media_destination is invalid
B_MEDIA_ALREADY_CONNE CTED	Can't connect nodes that are already connected
B_MEDIA_NOT_CONNECTED	There's no connection in place
B_MEDIA_BAD_CLIP_FORMA T	The clip format isn't valid
B_MEDIA_ADDON_FAILED	An add-on failed
B_MEDIA_ADDON_DISABLED	The add-on is disabled
B_MEDIA_CHANGE_IN_PROG RESS	A change is in progress, preventing the requested action
B_MEDIA_STALE_CHANGE_C OUNT	The change count is stale
B_MEDIA_ADDON_RESTRICT ED	The add-on is restricted
B_MEDIA_NO_HANDLER	There's no handler available meeting the specified needs
B_MEDIA_DUPLICATE_FORM AT	The specified format already exists
B_MEDIA_REALTIME_DISABL ED	Real-time memory allocation is disabled
B_MEDIA_REALTIME_UNAVAI LABLE	Real-time memory allocation isn't available

Mail Errors

Code	Description
B_MAIL_NO_DAEMON	mail_daemon not running
B_MAIL_UNKNOWN_USER	User unknown
B_MAIL_WRONG_PASSWOR D	Password doesn't match user
B_MAIL_UNKNOWN_HOST	POP or SMTP host unrecognized
B_MAIL_ACCESS_ERROR	Couldn't access host
B_MAIL_UNKNOWN_FIELD	Unrecognized message field name
B_MAIL_NO_RECIPIENT	Return to sender, address unknown
B_MAIL_INVALID_MAIL	Invalid mail invalid mail invalid

Device Kit Errors

Code
B_DEV_INVALID_IOCTL
B_DEV_NO_MEMORY
B_DEV_BAD_DRIVE_NUM
B_DEV_NO_MEDIA
B_DEV_UNREADABLE
B_DEV_FORMAT_ERROR
B_DEV_TIMEOUT
B_DEV_RECALIBRATE_ERROR
B_DEV_SEEK_ERROR
B_DEV_ID_ERROR
B_DEV_READ_ERROR
B_DEV_WRITE_ERROR
B_DEV_NOT_READY
B_DEV_MEDIA_CHANGE_REQUESTED
B_DEV_RESOURCE_CONFLICT
B_DEV_CONFIGURATION_ERROR
B_DEV_DISABLED_BY_USER
B_DEV_DOOR_OPEN

The Be Book The Support Kit The Support Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Threads and Teams

Declared in: be/kernel/OS.hheaders/be/kernel/OS.h
Library: libroot.so
Summary: *more...*

A thread is a synchronous process that executes a series of program instructions. When you launch an application, an initial thread—the *main thread*—is automatically created (or *spawned*) and told to run. From the main thread you can spawn and run additional threads; from each of these threads you can spawn and run more threads, and so on. The collection of threads that are spawned from the main thread—in other words, the threads that comprise an application—is called a team. All the threads in all teams run concurrently and asynchronously with each other.

For more information on threads and teams, see "Thread and Team Concepts".

Thread and Team Functions

estimate_max_scheduling_latency()

Declared in: be/kernel/scheduler.h

bigtime_t **estimate_max_scheduling_latency(**thread_id *thread* = -1**)**

Returns the scheduling latency, in microseconds, of the specified thread. Specify a `thread_id` of -1 to return the scheduling latency of the current thread.

exit_thread() , kill_thread() , kill_team() , on_exit_thread()

void **exit_thread(**status_t *return_value***)**
status_t **kill_thread(t**hread_id *thread***)**
status_t **kill_team(**team_id *team***)**
status_t **on_exit_thread(**void (**callback*)(void *), void **data***)**

These functions command one or more threads to halt execution:

- `exit_thread()` tells the calling thread to exit with a return value as given by the argument. Declaring the return value is only useful if some other thread is sitting in a `wait_for_thread()` call on this thread. `exit_thread()` sends a signal to the thread (after caching the return value in a known place).
- `kill_thread()` kills the thread given by the argument. The value that the thread will return to `wait_for_thread()` is undefined and can't be relied upon. `kill_thread()` is the same as sending a **SIGKILLTHR** signal to the thread.
- `kill_team()` kills all the threads within the given team. Again, the threads' return values are random. `kill_team()` is the same as sending a **SIGKILL** signal to any thread in the team. Each of the threads in the team is then handed a **SIGKILLTHR** signal.
- `on_exit_thread()` sets up the specified *callback* to be executed when the calling thread exits. The callback will receive the pointer *data* as an input argument.

Exiting a thread is a fairly safe thing to do—since a thread can only exit itself, it's assumed that the thread knows what it's doing. Killing some other thread or an entire team is a bit more drastic since the death certificate(s) will be delivered at an indeterminate time. In addition, killing a thread can leak memory since resources that were allocated by the thread may not be freed. Killing an entire team, on the other hand, won't leak since the system reclaims all resources when the team dies.

Keep in mind that threads die automatically (and their resources are reclaimed) if they're allowed to exit naturally. You should only need to kill a thread if something has gone screwy.

RETURN CODES

- `B_OK`. The thread or team was successfully killed.
- `B_BAD_THREAD_ID`. Invalid *thread* value.
- `B_BAD_TEAM_ID`. Invalid *team* value.
- `B_NO_MEMORY`. Returned by `on_exit_thread()` if there's no memory to construct the internal callback record.

---

find_thread()

```
thread_id find_thread(const char *name)
```

Finds and returns the thread with the given name. A *name* argument of `NULL` returns the calling thread.

A thread's name is assigned when the thread is spawned. The name can be changed thereafter through the `rename_thread()` function. Keep in mind that thread names needn't be unique: If two (or more) threads boast the same name, a `find_thread()` call on that name returns the first so-named thread that it finds. There's no way to iterate through identically-named threads.

RETURN CODES

- `B_NAME_NOT_FOUND`. *name* doesn't identify a valid thread.

---

get_team_info() , get_next_team_info()

```
status_t get_team_info(team_id team, team_info *info)
status_t get_next_team_info(int32 *cookie, team_info *info)
```

The functions copy, into the *info* argument, the `team_info` structure for a particular team. The `get_team_info()` function retrieves information for the team identified by *team*. For information about the kernel, use `B_SYSTEM_TEAM` as the *team* argument.

The `get_next_team_info()` version lets you step through the list of all teams. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns `B_BAD_VALUE` when there are no more areas to visit:

```
/* Get the team_info for every team. */
team_info info;
int32 cookie = 0;
```

```
while (get_next_team_info(0, &cookie, &info) == B_OK)
   ...
```

See team_info for a description of that structure.

RETURN CODES

- B_OK. The desired team information was found.
- B_BAD_TEAM_ID. *team* doesn't identify an existing team, or there are no more areas to visit.

---

get_thread_info() , get_next_thread_info()

```
status_t get_thread_info(thread_id thread, thread_info *info)
status_t  get_next_thread_info(team_id team,
    int32 *cookie,
    thread_info *info)
```

These functions copy, into the *info* argument, the thread_info structure for a particular thread:

The get_thread_info() function gets the information for the thread identified by *thread*.

The get_next_thread_info() function lets you step through the list of a team's threads through iterated calls. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns B_BAD_VALUE when there are no more threads to visit:

```
/* Get the thread_info for every thread in this team. */
thread_info info;
int32 cookie = 0;

while (get_next_thread_info(0, &cookie, &info) == B_OK)
   ...
```

The value of the **priority** field describes the thread's "urgency"; the higher the value, the more urgent the thread. The more urgent the thread, the more attention it gets from the CPU. Expected priority values fall between 0 and 120. See "Thread Priorities" for the full story.

Thread info is provided primarily as a debugging aid. None of the values that you find in a thread_info structure are guaranteed to be valid—the thread's state, for example, will almost certainly have changed by the time get_thread_info() returns.

RETURN CODES

- B_OK. The thread was found; *info* contains valid information.

- B_BAD_VALUE. *thread* doesn't identify an existing thread, *team* doesn't identify an existing team, or there are no more threads to visit.

---

has_data() see send_data()

---

rename_thread()

```
status_t rename_thread(thread_id thread, const char *name)
```

Changes the name of the given thread to *name*. The name can be no longer than B_OS_NAME_LENGTH (32 characters).

RETURN CODES

- B_OK. The thread was successfully named.
- B_BAD_THREAD_ID. *thread* argument isn't a valid thread_id number.

---

resume_thread()

```
status_t resume_thread(thread_id thread)
```

Tells a new or suspended thread to begin executing instructions. If the thread has just been spawned, it enters and executes the thread function declared in spawn_thread(). If the thread was previously suspended (through suspend_thread()), it continues from where it was suspended.

You can't use this function to wake up a sleeping thread, or to unblock a thread that's waiting to acquire a semaphore or waiting in a receive_data() call. However, you can unblock any of these threads by suspending and *then* resuming. Blocked threads that are resumed return B_INTERRUPTED.

resume_thread() is the same as sending a **SIGCONT** signal to the thread.

RETURN CODES

- B_OK. The thread was successfully resumed.
- B_BAD_THREAD_ID. *thread* argument isn't a valid thread_id number.
- B_BAD_THREAD_STATE. The thread isn't suspended.

---

receive_data()

Retrieves a message from the thread's message cache. The message will have been placed there through a previous send_data() function call. If the cache is empty, receive_data() blocks until one shows up—it never returns empty-handed.

The thread_id of the thread that called send_data() is returned by reference in the *sender* argument. Note that there's no guarantee that the sender will still be alive by the time you get its ID. Also, the value of *sender* going into the function is ignored—you can't ask for a message from a particular sender.

The send_data() function copies two pieces of data into a thread's message cache:

- A single four-byte code that's delivered as receive_data()'s return value,
- and an arbitrarily long data buffer that's copied into receive_data()'s *buffer* argument (you must allocate and free *buffer* yourself). The *buffer_size* argument tells the function how many bytes of data to copy. If you don't need the data buffer—if the code value returned directly by the function is sufficient—you set *buffer* to **NULL** and *buffer_size* to 0.

Unfortunately, there's no way to tell how much data is in the cache before you call receive_data():

- If there's more data than *buffer* can accommodate, the unaccommodated portion is discarded—a second receive_data() call will not read the rest of the message.
- Conversely, if receive_data() asks for more data than was sent, the function returns with the excess portion of *buffer* unmodified—receive_data() doesn't wait for another send_data() call to provide more data with which to fill up the buffer.

Each **receive_data()** corresponds to exactly one send_data(). Lacking a previous invocation of its mate, receive_data() will block until send_data() is called. If you don't want to block, you should call has_data() before calling receive_data() (and proceed to receive_data() only if has_data() returns **true**).

RETURN CODES

- If successful, the function returns the message's four-byte code.
- B_INTERRUPTED. A blocked receive_data() call was interrupted by a signal.

---

send_data() , receive_data() , has_data()

status_t **send_data(thread_id** *thread*, int32 *code*, void **buffer*, **size_t** *buffer_size***)**
int32 **receive_data(thread_id** **sender*, void **buffer*, **size_t** *buffer_size***)**
bool **has_data(thread_id** *thread***)**

Every thread has a one-message-deep message cache associated with it. These functions access that cache.

send_data() copies a message into *thread*'s message cache. The target thread retrieves the message (and empties the cache) by calling **receive_data()**.

There are two parts to the message:

- A single four-byte *code* passed as an argument to **send_data()** and returned directly by **retrieve_data()**.
- A *buffer* of data that's *buffer_size* bytes long (*buffer* can be **NULL**, in which case *buffer_size* should be 0). The data is copied into the target thread's cache, and then copied into **receive_data()**'s *buffer* (which must be allocated). The calling threads retain responsibility for freeing their buffers.

In addition to returning the *code* directly, and copying the message data into its *buffer* argument, **receive_data()** sets *sender* to the id of the thread that sent the message.

send_data() blocks if there's an unread message in the target thread's cache; otherwise it returns immediately (i.e. it doesn't wait for the target to call **receive_data()**). Analogously, **receive_data()** blocks until there's a message to retrieve.

In the following example, the main thread spawns a thread, sends it a message, and then tells the thread to run:

```
main()
{
   thread_id other_thread;
   int32 code = 63;
   char *buf = "Hello";

   other_thread = spawn_thread(thread_func, ...);
   send_data(other_thread, code, (void *)buf, strlen(buf));
   resume_thread(other_thread);
   ...
}
```

To retrieve the message, the target thread calls **receive_data()**:

```
int32 thread_func(void *data)
{
   thread_id sender;
   int32 code;
   char buf[512];

   code = receive_data(&sender, (void *)buf, sizeof(buf));
   ...
}
```

Keep in mind that the message data is *copied* into the buffer; you must allocate adequate storage for the data. If the buffer isn't big enough to accommodate all the datain the

message, the left-over portion is thrown away. Note, however, that there isn't any way for a thread to determine how much data has been copied into its message cache.

has_data() returns **true** if *thread* has a message in its message cache. Ostensibly, you use this function before calling **send_data()** or **receive_data()** to avoid blocking:

```
if (!has_data(target_thread))
   err = send_data(target_thread, ...);

/* or */

if (has_data(find_thread(NULL))
   code = receive_data(...);
```

This works for **receive_data()**, but notice that there's a race condition between the **has_data()** and **send_data()** calls. Another thread could send a message to the target in the interim.

RETURN CODES

send_data() returns:

- **B_OK**. The data was successfuly sent.
- **B_BAD_THREAD_ID**. *thread* doesn't identify a valid thread.
- **B_NO_MEMORY**. The target couldn't allocate enough memory for its copy of *buffer*.
- **B_INTERRUPTED**. The function blocked, but a signal unblocked it.

---

set_thread_priority() , suggest_thread_priority()

```
status_t set_thread_priority(thread_id thread, int32 new_priority)
```

Declared in: be/kernel/scheduler.h

```
int32 suggest_thread_priority(uint32 what = B_DEFAULT_MEDIA_PRIORITY,
    int32 period = 0,
    bigtime_t jitter = 0,
    bigtime_t length = 0)
```

set_thread_priority() resets the given thread's priority to *new_priority*. The priority is expected to be between 0 and 120. See "Thread Priorities" for a description of the priority scheme, and "Thread Priority Values" for a list of pre-defined priority constants.

suggest_thread_priority() takes information about a thread and returns a suggested priority that you can pass to **set_thread_priority()** (or, more likely, to **spawn_thread()**).

The *what* value is a bit mask that indicates the type of activities the thread will be used for. The possible values are listed in Suggested Thread Priorities.

period is the number of times per second the thread needs to be run (specify 0 if it needs to run continuously). *jitter* is an estimate, in microseconds, of how much the period can vary as long as the average stays at *period* times per second.

length is an approximation of the amount of time, in microseconds, the thread will typically run per invocation (i.e., the amount of time that will pass between the moment it receives a message, through processing it, until it's again waiting for another message).

For example, if you're spawning a thread to handle video refresh for a computer game, and you want the display to update 30 times per second, you might use code similar to the following:

```
int32 priority;
priority = suggest_thread_priority(B_LIVE_3D_RENDERING, 30, 1000, 150);
th = spawn_thread(func, "render_thread", priority, NULL)
```

This spawns the rendering thread with a priority appropriate for a thread for live 3D rendering which wants to be run 30 times per second, with a variation of only 1000 microseconds. Each invocation of the thread's code is estimated to take 150 microseconds. Obviously the *jitter* and *length* values would have to be tuned to the particular application.

RETURN CODES

`set_thread_priority()` returns...

Positive integers. If the function is successful, the previous priority is returned.

- `B_BAD_THREAD_ID`. *thread* doesn't identify a valid thread.

---

snooze() , snooze_until()

status_t **snooze(**bigtime_t *microseconds***)**
status_t **snooze_until(**bigtime_t *microseconds*, int *timebase***)**

`snooze()` blocks the calling thread for the given number of microseconds.

`snooze_until()` blocks until an absolute time measured in the given timebase. Currently, the only allowed value for timebase is `B_SYSTEM_TIMEBASE`, which measures time against the system clock (as reported by `system_time()`).

RETURN CODES

- `B_OK`. The thread went to sleep and is now awake.
- `B_INTERRUPTED`. The thread received a signal while it was sleeping.

---

snooze_until() see snooze()

---

spawn_thread()

```
thread_id spawn_thread(thread_func func,
    const char *name,
    int32 priority,
    void *data)
```

Creates a new thread and returns its `thread_id` identifier (a positive integer). The arguments are:

- *func* is a pointer to a thread function. This is the function that the thread will execute when it's told to run. See "The Thread Function" for details.
- *name* is the name that you wish to give the thread. It can be, at most, `B_OS_NAME_LENGTH` (32) characters long.
- *priority* is the CPU priority level of the thread. This value should be between 0 and 120; he higher the priority, the more attention the thread gets. See Thread Priorities for a description of the priorities, and Thread Priority Values for a list of priority constants.
- *data* is forwarded as the argument to the thread function.

A newly spawned thread is in a suspended state (`B_THREAD_SUSPENDED`). To tell the thread to run, you pass its `thread_id` to the `resume_thread()` function. The thread will continue to run until the thread function exits, or until the thread is explicitly killed (through a signal or a call to `exit_thread()`, `kill_thread()`, or `kill_team()`).

RETURN CODES

- `B_NO_MORE_THREADS`. All `thread_id` numbers are currently in use.
- `B_NO_MEMORY`. Not enough memory to allocate the resources for another thread.

---

suggest_thread_priority() see set_thread_priority()

---

suspend_thread()

```
status_t suspend_thread(thread_id thread)
```

Halts the execution of the given thread, but doesn't kill the thread entirely. The thread remains suspended (`suspend_thread()` blocks) until it's told to run through the `resume_thread()` function. Nothing prevents you from suspending your own thread, i.e.:

```
suspend_thread(find_thread(NULL));
```

Of course, this is only smart if you have some other thread that will resume you later.

You can suspend any thread, regardless of its current state. But be careful: If the thread is blocked on a semaphore (for example), the subsequent `resume_thread()` call will "hop over" the semaphore acquisition.

Suspensions don't nest. A single `resume_thread()` unsuspends a thread regardless of the number of `suspend_thread()` calls it has received.

`suspend_thread()` is the same as sending a **`SIGSTOP`** signal to the thread.

RETURN CODES

- `B_OK`. The thread is now suspended.
- `B_BAD_THREAD_ID`. *thread* isn't a valid `thread_id` number.

---

wait_for_thread()

status_t **wait_for_thread**(thread_id *thread*, status_t **exit_value*)

This function causes the calling thread to wait until *thread* (the "target thread") has died. If *thread* is suspended (or freshly spawned), `wait_for_thread()` will resume it.

When the target thread is dead, the value that was returned by its thread function (or imposed by `exit_thread()`) is returned in *exit_value*. If the target thread was killed (by `kill_thread()` or `kill_team()`), or if the thread function doesn't return a value, the value returned in *exit_value* will be unreliable.

You must pass a valid pointer as the second argument to wait_for_thread(). You mustn't pass **`NULL`** even if you're not interested in the return value.

RETURN CODES

- `B_OK`. The target is now dead.
- `B_BAD_THREAD_ID`. *thread* isn't a valid `thread_id` number.
- `B_INTERRUPTED`. The target was killed by a signal. This includes `kill_thread()`, `kill_team()`, and `exit_thread()`.

Thread and Team Structures and Types

team_id , thread_id

typedef int32 **team_id** ;

typedef int32 **thread_id** ;

These id numbers uniquely identify teams and threads, respecitvely.

team_info

```
typedef struct {
    team_id team;
    int32 thread_count;
    int32 image_count;
    int32 area_count;
    thread_id debugger_nub_thread;
    port_id debugger_nub_port;
    int32 argc;
    char args[64];
    uid_t uid;
    gid_t gid;
    } team_info;
```

The **team_info** structure returns information about a team. To retrieve one of these structures, use **get_team_info()** or **get_next_team_info()**.

The first field is obvious; the next three reasonably so: They give the number of threads that have been spawned, images that have been loaded, and areas that have been created or cloned within this team.

The debugger fields are used by the, uhm, the...debugger?

The **argc** field is the number of command line arguments that were used to launch the team; **args** is a copy of the first 64 characters from the command line invocation. If this team is an application that was launched through the user interface (by double-clicking, or by accepting a dropped icon), then **argc** is 1 and **args** is the name of the application's executable file.

uid and **gid** identify the user and group that "owns" the team. You can use these values to play permission games.

thread_func

```
typedef int32 (*thread_func)(void *data);
```

thread_func is the prototype for a thread's *thread function*. You specify a thread function by passing a **thread_func** as the first argument to **spawn_thread()**; the last argument to **spawn_thread()** is forwarded as the thread function's *data* argument. When the thread function exits, the spawned thread is automatically killed. To retrieve a **thread_func**'s return value, some other thread must be waiting in a **wait_for_thread()** call.

Note that **spawn_thread()** *doesn't* copy the data that *data* points to. It simply passes the pointer through literally. Never pass a pointer that's allocated locally (on the stack).

---

thread_info

```
typedef struct {
    thread_id thread;
    team_id team;
    char  name[B_OS_NAME_LENGTH];
    thread_state state;
    sem_id sem;
    int32 priority;
    bigtime_t user_time;
    bigtime_t kernel_time;
    void *stack_base;
    void *stack_end;
    } thread_info
```

The thread_info structure contains information about a thread. To retrieve one of these structure, use **get_thread_info()** or **get_next_thread_info()**.

The **thread**, **team**, and **name** fields contain the indicated information.

state describes what the thread is currently doing (see **thread_state** for the list of states). If the thread is waiting to acquire a semaphore, **sem** is that semaphore.

priority is a value that indicates the level of attention the thread gets (see Thread Priority).

user_time and **kernel_time** are the amounts of time, in microseconds, the thread has spent executing user code and the amount of time the kernel has run on the thread's behalf, respectively.

stack_base and **stack_end** are pointers to the first byte and last bytes in the thread's execution stack. Currently, the stack size is fixed at around 256k.

The two stack pointers are currently inverted such that **stack_base** is *less* than **stack_end**. (In a stack-grows-down world, the base should be greater than the end.)

Thread and Team Constants

B_SYSTEM_TEAM

```
#define B_SYSTEM_TEAM ...
```

Use this constant as the first argument to `get_team_info()` to get team information about the kernel).

B_SYSTEM_TIMEBASE

```
#define B_SYSTEM_TIMEBASE ...
```

The system timebase constant is used as a basis for time measurement in the `snooze_until()` function. (Currently, it's the only timebase available.)

be_task_flags

Declared in: be/kernel/scheduler.h

```
enum be_task_flags { B_DEFAULT_MEDIA_PRIORITY,
    B_OFFLINE_PROCESSING,
     B_STATUS_RENDERING,
    B_USER_INPUT_HANDLING
        B_LIVE_VIDEO_MANIPULATION
    };
```

Constant	Meaning
B_DEFAULT_MEDIA_PRIORITY	The thread isn't doing anything specialized.
B_OFFLINE_PROCESSING	The thread is doing non-real-time computations.
B_STATUS_RENDERING	The thread is rendering a status or preview display.
B_USER_INPUT_HANDLING	The thread is handling user input.
B_LIVE_VIDEO_MANIPULATION	The thread is processing live video (filtering, compression, decompression, etc.).
B_VIDEO_PLAYBACK	The thread is playing back video from a hardware device.
B_VIDEO_RECORDING	The thread is recording video from a hardware device.
B_LIVE_AUDIO_MANIPULATION	The thread is doing real-time manipulation of live audio data (filtering, compression, decompression, etc.).
B_AUDIO_PLAYBACK	The thread is playing back audio from a hardware device.

B_AUDIO_RECORDING	The thread is recording audio from a hardware device.
B_LIVE_3D_RENDERING	The thread is performing live 3D rendering.
B_NUMBER_CRUNCHING	The thread is doing data processing.

These constants describe what the thread is designed to do. You use these constants when asking for a suggested priority (see **suggest_thread_priority()**).

These constants may not be used as actual thread priority values—do not pass one of these values as the *priority* argument to **spawn_thread()**.

Thread Priority Values

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORIT Y	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	**100**
B_URGENT_PRIORITY	**110**
B_REAL_TIME_PRIORITY	**120**

The thread priority values are used to set the "urgency" of a thread. Although you can reset a thread's priority through **set_thread_priority()**, the priority is initially—and almost always permanently—set in **spawn_thread()**. As shown here, there are two types of

---

thread_state

enum { ... } **thread_state**

State	Meaning
B_THREAD_RUNNING	The thread is currently receiving attention from a CPU.
B_THREAD_READY	The thread is waiting for its turn to receive attention.
B_THREAD_SUSPENDED	The thread has been suspended or is freshly-spawned and is waiting to start.
B_THREAD_WAITING	The thread is waiting to acquire a semaphore. The **sem** field of the thread's **thread_info** structure will tell you which semaphore.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** function call.
B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.

A thread's state tells you what the thread is currently doing. To get the state, look in the **state** field of the **thread_info** structure (retrieved through **get_thread_info()**).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Kernel Kit Table of Contents

- The Kernel Kit
- Threads and Teams
- Thread and Team Concepts
- Semaphores
- Semaphore Concepts
- Semaphore Examples
- Ports
- Port Concepts
- Areas
- Areas Concepts
- Area Examples
- Images
- Image Concepts
- System Information
- Time Information
- Miscellaneous Functions and Constants

The Be Book **The Kernel Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Kernel Kit: Master Index

A

acquire_sem()	Semaphores
acquire_sem_etc()	Semaphores
B_ADD_ON_IMAGE	Images
B_APP_IMAGE	Images
Area Examples	Area Examples
Area Examples	Area Examples
Area Functions	Areas
Area IDs and Area Names	Areas Concepts
Area Info	Areas Concepts
area_for()	Areas
area_info	Areas
Areas	Areas
Areas	Areas
Areas Concepts	Areas Concepts
Areas Concepts	Areas Concepts
B_AUDIO_PLAYBACK	Threads and Teams
B_AUDIO_RECORDING	Threads and Teams

B

be_task_flags	Threads and Teams

boot_time	System Information
bus_clock_speed	System Information

C

capacity	Ports
clear_caches()	Miscellaneous Functions and Constants
clone_area()	Areas
Cloned Memory	Areas Concepts
close_port()	Ports
Constants	Miscellaneous Functions and Constants
Controlling a Thread	Thread and Team Concepts
count	Semaphores
cpu_clock_speed	System Information
cpu_count	System Information
cpu_info	System Information
cpu_infos	System Information
cpu_revision	System Information
cpu_type	System Information
B_CPU_X86_VENDOR_MASK	System Information
create_area()	Areas
create_port()	Ports
create_sem()	Semaphores
Creating a Shared Library	Image Concepts

Creating and Destroying a Port	Port Concepts
Creating and Using an Add-on Image	Image Concepts

D

Death and the Main Thread	Thread and Team Concepts
debugger()	Miscellaneous Functions and Constants
B_DEFAULT_MEDIA_PRIORITY	Threads and Teams
delete_area()	Areas
delete_port()	Ports
delete_sem()	Semaphores
Deleting a Semaphore	Semaphore Concepts
Deleting an Area	Areas Concepts
disable_debugger()	Miscellaneous Functions and Constants
B_DISPLAY_PRIORITY	Threads and Teams

E

estimate_max_scheduling_latency()	Threads and Teams
Example 1: Creating and Writing into an Area	Area Examples
Example 2: Reading a File into an Area	Area Examples

Example 3: Accessing a Designated Area	Area Examples
Example 4: Cloning and Sharing an Area	Area Examples
Example 5: Cloning Addresses	Area Examples
exit_thread()	Threads and Teams
Exporting Add-on Symbols	Image Concepts
Exporting and Importing Symbols	Image Concepts

F

find_area()	Areas
find_port()	Ports
find_thread()	Threads and Teams
Functions	Miscellaneous Functions and Constants

G

get_area_info()	Areas
get_image_info()	Images
get_image_symbol()	Images
get_next_area_info()	Areas
get_next_image_info()	Images
get_next_port_info()	Ports
get_next_sem_info()	Semaphores
get_next_team_info()	Threads and Teams
get_next_thread_info()	Threads and Teams

get_nth_image_symbol()	Images
get_port_info()	Ports
get_sem_count()	Semaphores
get_sem_info()	Semaphores
get_system_info()	System Information
get_team_info()	Threads and Teams
get_thread_info()	Threads and Teams

H

has_data()	Threads and Teams

I

id	System Information
Image Concepts	Image Concepts
Image Concepts	Image Concepts
Image Functions	Images
image_info	Images
image_type	Images
Images	Images
Images	Images
B_INFINITE_TIMEOUT	Miscellaneous Functions and Constants
Inter-application Semaphores	Semaphore Concepts
is_computer_on()	System Information

is_computer_on_fire()	System Information

K

The Kernel Kit	The Kernel Kit
The Kernel Kit	The Kernel Kit
kernel_build_date	System Information
kernel_build_time	System Information
kernel_name	System Information
kernel_version	System Information
kill_team()	Threads and Teams
kill_thread()	Threads and Teams

L

lastest_holder	Semaphores
latest_holder	Semaphores
B_LIBRARY_IMAGE	Images
B_LIVE_3D_RENDERING	Threads and Teams
B_LIVE_AUDIO_MANIPULATIO	Threads and Teams
B_LIVE_VIDEO_MANIPULATIO	Threads and Teams
load_add_on()	Images
load_image()	Images
Loading an Add-on Image	Image Concepts
Loading an App Image	Image Concepts
Locking an Area	Areas Concepts

B_LOW_PRIORITY	Threads and Teams

M

machine_id	System Information
B_MAX_CPU_COUNT	System Information
max_resource	System Information
The Message Queue: Reading and Writing Port Messages	Port Concepts
Miscellaneous Functions and Constants	Miscellaneous Functions and Constants
Miscellaneous Functions and Constants	Miscellaneous Functions and Constants

N

name	Ports
name	Semaphores
B_NORMAL_PRIORITY	Threads and Teams
B_NUMBER_CRUNCHING	Threads and Teams

O

B_OFFLINE_PROCESSING	Threads and Teams
on_exit_thread()	Threads and Teams
B_OS_NAME_LENGTH	Miscellaneous Functions and Constants

P

page_faults	System Information
B_PAGE_SIZE	Miscellaneous Functions and Constants
Passing Data to a Thread	Thread and Team Concepts
platform_type	System Information
Port Concepts	Port Concepts
Port Concepts	Port Concepts
Port Functions	Ports
Port Messages	Port Concepts
port	Ports
Port Structures and Constants	Ports
port_buffer_size()	Ports
port_buffer_size_etc()	Ports
port_count()	Ports
port_info	Ports
Ports	Ports
Ports	Ports

Q

queue_count	Ports

R

read_port()	Ports
read_port_etc()	Ports
real_time_clock()	Time Information
real_time_clock_usecs()	Time Information
B_REAL_TIME_DISPLAY_PRIORITY	Threads and Teams
B_REAL_TIME_PRIORITY	Threads and Teams
receive_data()	Threads and Teams
release_sem()	Semaphores
release_sem_etc()	Semaphores
rename_thread()	Threads and Teams
resize_area()	Areas
resume_thread()	Threads and Teams

S

sem	Semaphores
sem_id	Semaphores
sem_info	Semaphores
Semaphore Concepts	Semaphore Concepts
Semaphore Concepts	Semaphore Concepts
Semaphore Constants	Semaphores
Semaphore Control Flags	Semaphores
Semaphore Example 1: Locking	Semaphore Examples
Semaphore Example 2: Benaphores	Semaphore Examples
Semaphore Example 3: Imposing an Execution Order	Semaphore Examples

Semaphore Examples	Semaphore Examples
Semaphore Examples	Semaphore Examples
Semaphore Functions	Semaphores
Semaphore Structures and Types	Semaphores
Semaphores	Semaphores
Semaphores	Semaphores
send_data()	Threads and Teams
set_alarm()	Miscellaneous Functions and Constants
set_area_protection()	Areas
set_port_owner()	Ports
set_real_time_clock()	Time Information
set_sem_owner()	Semaphores
set_signal_stack()	Miscellaneous Functions and Constants
set_thread_priority()	Threads and Teams
Sharing an Area Between Applications	Areas Concepts
snooze()	Threads and Teams
snooze_until()	Threads and Teams
spawn_thread()	Threads and Teams
Spawning a Thread	Thread and Team Concepts
B_STATUS_RENDERING	Threads and Teams
suggest_thread_priority()	Threads and Teams
suspend_thread()	Threads and Teams
Symbols	Image Concepts

Synchronizing Threads	Thread and Team Concepts
System Info Functions	System Information
System Info Structures and Constants	System Information
System Information	System Information
System Information	System Information
system_info	System Information
B_SYSTEM_TEAM	Threads and Teams
system_time()	Time Information
B_SYSTEM_TIMEBASE	Threads and Teams

T

team	Ports
team	Semaphores
team_id	Threads and Teams
team_info	Threads and Teams
Telling a Thread to Run	Thread and Team Concepts
Thread and Team Concepts	Thread and Team Concepts
Thread and Team Concepts	Thread and Team Concepts
Thread and Team Constants	Threads and Teams
Thread and Team Functions	Threads and Teams
Thread and Team Structures and	Threads and Teams

Types	
The Thread Count	Semaphore Concepts
The Thread Function	Thread and Team Concepts
Thread Names	Thread and Team Concepts
Thread Priorities	Thread and Team Concepts
Thread Priority Values	Threads and Teams
The Thread Queue	Semaphore Concepts
B_THREAD_ASLEEP	Threads and Teams
thread_func	Threads and Teams
thread_id	Threads and Teams
thread_info	Threads and Teams
B_THREAD_READY	Threads and Teams
B_THREAD_RECEIVING	Threads and Teams
B_THREAD_RUNNING	Threads and Teams
thread_state	Threads and Teams
B_THREAD_SUSPENDED	Threads and Teams
B_THREAD_WAITING	Threads and Teams
Threads and App Images	Thread and Team Concepts
Threads and Teams	Threads and Teams
Threads and Teams	Threads and Teams
Time Functions	Time Information
Time Information	Time Information
Time Information	Time Information

total_count	Ports

U

unload_add_on()	Images
B_URGENT_DISPLAY_PRIORIT	Threads and Teams
B_URGENT_PRIORITY	Threads and Teams
B_USER_INPUT_HANDLING	Threads and Teams

V

B_VIDEO_PLAYBACK	Threads and Teams
B_VIDEO_RECORDING	Threads and Teams

W

wait_for_thread()	Threads and Teams
write_port()	Ports
write_port_etc()	Ports

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Semaphores

Declared in: be/kernel/OS.h
Library: libroot.so
Summary: *more...*

A semaphore is a token that's used to synchronize multiple threads. The semaphore concept is simple: To enter into a semaphore-protected "critical section", a thread must first "acquire" the semaphore, through the **`acquire_sem()`** function. When it passes out of the critical section, the thread "releases" the semaphore through **`release_sem()`**.

The advantage of the semaphore system is that if a thread can't acquire a semaphore (because the semaphore is yet to be released by the previous acquirer), the thread blocks in the **`acquire_sem()`** call. While it's blocked, the thread doesn't waste any cycles.

For the full story about semaphores, see "Semaphore Concepts". For some code examples, see "Semaphore Examples".

Semaphore Functions

acquire_sem() , acquire_sem_etc()

```
status_t acquire_sem(sem_id sem)
status_t acquire_sem_etc(sem_id sem,
    uint32 count,
    uint32 flags,
    bigtime_t timeout)
```

These functions attempt to acquire the semaphore identified by the *sem* argument. Except in the case of an error, `acquire_sem()` doesn't return until the semaphore has actually been acquired.

`acquire_sem_etc()` is the full-blown acquisition version: It's essentially the same as `acquire_sem()`, but, in addition, it lets you acquire a semaphore more than once, and also provides a timeout facility:

- The *count* argument lets you specify that you want the semaphore to be acquired *count* times. This means that the semaphore's thread count is decremented by the specified amount. It's illegal to specify a count that's less than 1.
- To enable the timeout, you add **B_ABSOLUTE_TIMEOUT** or **B_RELATIVE_TIMEOUT** to the *flags* argument. *timeout* to the amount of time, in microseconds, that you're willing to wait, measured relative to now (relative timeout), or in comparison to the value returned by system_time() (absolute timeout). The function returns **B_TIMED_OUT** if the semaphore isn't acquired within the specified time. If you specify a relative *timeout* of 0 and the semaphore isn't immediately available, the function immediately returns **B_WOULD_BLOCK**.

The Kernel Kit defines two other semaphore-acquisition flag constants (**B_CAN_INTERRUPT** and **B_CHECK_PERMISSION**). These additional flags are used by device drivers—adding these flags into a "normal" (or "user-level") acquisition has no effect. However, you should be aware that the **B_CHECK_PERMISSION** flag is always added in to user-level semaphore acquisition in order to protect system-defined semaphores.

Other than the timeout and the acquisition count, there's no difference between the two acquisition functions. Specifically, any semaphore can be acquired through either of these functions; you always release a semaphore through `release_sem()` (or `release_sem_etc()`) regardless of which function you used to acquire it.

To determine if the semaphore is available, the function looks at the semaphore's thread count (before decrementing it):

- If the thread count is positive, the semaphore is available and the current acquisition succeeds. The `acquire_sem()` (or `acquire_sem_etc()`) function returns immediately upon acquisition.

- If the thread count is zero or less, the calling thread is placed in the semaphore's thread queue where it waits for a corresponding `release_sem()` call to de-queue it (or for the timeout to expire).

RETURN CODES

- `B_NO_ERROR`. The semaphore was successfully acquired.
- `B_BAD_SEM_ID`. The *sem* argument doesn't identify a valid semaphore. It's possible for a semaphore to become invalid while an acquisitive thread is waiting in the semaphore's queue. For example, if your thread calls `acquire_sem()` on a valid (but unavailable) semaphore, and then some other thread deletes the semaphore, your thread will return `B_BAD_SEM_ID` from its call to `acquire_sem()`.
- `B_INTERRUPTED`. The acquisition was interrupted by a signal. In this case, the semaphore has *not* been acquired.

The other return values apply to `acquire_sem_etc()` only:

- `B_BAD_VALUE`. Illegal *count* value (less than 1).
- `B_WOULD_BLOCK`. You specified a relative *timeout* of 0 and the semaphore isn't available.
- `B_TIMED_OUT`. The timeout expired (for all values of *timeout* other than 0).

---

create_sem()

sem_id **create_sem(**uint32 *thread_count*, const char **name***)**

Creates a new semaphore and returns a system-wide `sem_id` number that identifies it. The arguments are:

- *thread_count* initializes the semaphore's *thread count*, the counting variable that's decremented and incremented as the semaphore is acquired and released (respectively). You can pass any non-negative number as the count, but you typically pass either 1 or 0.
- *name* is an optional string name that you can assign to the semaphore. The name is meant to be used only for debugging. A semaphore's name needn't be unique—any number of semaphores can have the same name.

Valid `sem_id` numbers are positive integers. You should always check the validity of a new semaphore through a construction such as

```
if ((my_sem = create_sem(1,"My Semaphore")) < B_OK)
   /* If it's less than B_NO_ERROR, my_sem is invalid. */
```

`create_sem()` sets the new semaphore's owner to the team of the calling thread. Ownership may be re-assigned through the `set_sem_owner()` function. When the owner dies (when all the threads in the team are dead), the semaphore is automatically deleted. The owner is also

signficant in a **delete_sem()** call: Only those threads that belong to a semaphore's owner are allowed to delete that semaphore.

RETURN CODES

- **B_BAD_VALUE**. Invalid *thread_count* value (less than 0).
- **B_NO_MEMORY**. Not enough memory to allocate the semaphore's name.
- **B_NO_MORE_SEMS**. All valid **sem_id** numbers are being used.

---

delete_sem()

status_t **delete_sem(**sem_id *sem***)**

Deletes the semaphore identified by the argument. If there are any threads waiting in the semaphore's thread queue, they're immediately unblocked.

This function may only be called from a thread that belongs to the semaphore's owner.

RETURN CODES

- **B_NO_ERROR**. The semaphore was successfully deleted.
- **B_BAD_SEM_ID**. *sem* is invalid, or the calling thread doesn't belong to the team that owns the semaphore.

---

get_sem_count()

status_t **get_sem_count(**sem_id *sem*, int32 **thread_count***)**

For amusement purposes only; never predicate your code on this function.

Returns, by reference in *thread_count*, the value of the semaphore's thread count variable:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **acquire_sem()** calls will return without blocking.
- If the count is zero, there are no queued threads, but the next **acquire_sem()** call will block.
- A negative count (*-n*) means there are *n* threads in the semaphore's thread queue and the next call to **acquire_sem()** will block.

By the time this function returns and you get a chance to look at the *thread_count* value, the semaphore's thread count may have changed. Although watching the thread count might help you while you're debugging your program, this function shouldn't be an integral part of the design of your application.

RETURN CODES

- B_NO_ERROR. Success.
- B_BAD_SEM_ID. *sem* is invalid(*thread_count* isn't changed).

---

get_sem_info() , get_next_sem_info()

```
status_t get_sem_info(sem_id sem, sem_info *info)
```

```
status_t get_next_sem_info(team_id team,
    uint32 *cookie,
    sem_info *info)
```

Copies information about a particular semaphore into the `sem_info` structure designated by *info*. The first version of the function designates the sempahore directly, by `sem_id`.

The `get_next_sem_info()` version lets you step through the list of a team's semaphores through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns B_BAD_VALUE when there are no more sempahores to visit:

```
/* Get the sem_info for every sempahore in this team. */
sem_info info;
int32 cookie = 0;

while (get_next_sem_info(0, &cookie, &info) == B_OK)
   ...
```

RETURN CODES

- B_NO_ERROR. Success.
- B_BAD_SEM_ID. Invalid *sem* value.
- B_BAD_TEAM_ID. Invalid *team* value.

---

release_sem() , release_sem_etc()

```
status_t release_sem(sem_id sem)
```

```
status_t release_sem_etc(sem_id sem, int32 count, uint32 flags)
```

The `release_sem()` function de-queues the thread that's waiting at the head of the semaphore's thread queue (if any), and increments the semaphore's thread count. `release_sem_etc()` does the same, but for *count* threads.

Normally, releasing a semaphore automatically invokes the kernel's scheduler. In other words, when your thread calls `release_sem()`, you're pretty much guaranteed that some other thread will be switched in immediately afterwards, even if your thread hasn't gotten its fair share of CPU time. If you want to subvert this automatism, call `release_sem_etc()` with a *flags* value of **`B_DO_NOT_RESCHEDULE`**. Preventing the automatic rescheduling is particularly useful if you're releasing a number of different semaphores all in a row: By avoiding the rescheduling you can prevent some unnecessary context switching.

RETURN CODES

- **`B_NO_ERROR`**. The semaphore was successfully released.
- **`B_BAD_SEM_ID`**. Invalid *sem* value.
- **`B_BAD_VALUE`**. Invalid *count* value (less than zero; `release_sem_etc()` only).

See also: **`acquire_sem()`**

---

set_sem_owner()

`status_t set_sem_owner(sem_id` *sem*, **`team_id`** *team***`)`**

Transfers ownership of the designated semaphore to *team*. A semaphore can only be owned by one team at a time; by setting a semaphore's owner, you remove it from its current owner.

There are no restrictions on who can own a semaphore, or on who can transfer ownership. In practice, however, the only reason you should ever transfer ownership is if you're writing a device driver and you need to bequeath a semaphore to the kernel (the team of which is known, for this purpose, as **`B_SYSTEM_TEAM`**).

Semaphore ownership is meaningful for two reason:

- When a team dies (when all its threads are dead), the semaphores that are owned by that team are deleted.
- Threads can only by deleted by threads that belongs to a semaphore's owner.

To discover a semaphore's owner, use the **`get_sem_info()`** function.

RETURN CODES

- **`B_NO_ERROR`**. Ownership was successfully transferred.
- **`B_BAD_SEM_ID`**. Invalid *sem* value.
- **`B_BAD_TEAM_ID`**. Invalid *team* value.

Semaphore Structures and Types

sem_id

```
typedef int32 sem_id;
```

sem_id numbers identify semaphores. The id is assigned when the semaphore is created (`create_sem()`). The values are unique across the system.

sem_info

```
typedef struct sem_info {
    sem_id sem;
    team_id team;
    char name[B_OS_NAME_LENGTH];
    int32 count;
    thread_id latest_holder;
    }
```

The sem_info structure supplies information about a semaphore. You retrieve the structure through the `get_sem_info()` function. The information in the `sem_info` structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a semaphore as it exists just before the info-retrieving function returns.

The fields are:

- **sem** . The `sem_id` number of the semaphore.
- **team** . The `team_id` of the semaphore's owner.
- **name** . The name assigned to the semaphore.
- **count** . The semaphore's thread count.
- **latest_holder** . The thread that most recently acquired the semaphore.

The **lastest_holder** field is *highly* undependable; in some cases, the kernel doesn't even record the semaphore acquirer. Although you can use this field as a hint while debugging, you shouldn't take it too seriously. Love, Mom.

Semaphore Constants

Semaphore Control Flags

B_CAN_INTERRUPT Tells the kernel that the semaphore can be interrupted by a signal.
B_DO_NOT_RESCHEDULE Tells the scheduler not to run after a semaphore is released. In other words, the thread th released the semaphore gets to keep running.
B_CHECK_PERMISSION Makes sure that the semaphore acquirer/releaser is running at the proper level. This is alway into user-level acquisition and release.
B_RELATIVE_TIMEOUT Used to set a timeout that's relative to now.
B_ABSOLUTE_TIMEOUT Used to set a timeout that's measured against the system clock.
B_TIMEOUT Obsolete; use **B_RELATIVE_TIMEOUT**.

These constants are combined to form the *flag* argument to the `acquire_sem_etc()` and `release_sem_etc()` functions.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Semaphores Function Summary

Function Summary

Function	Description
acquire_sem()	Acquires a semaphore
acquire_sem_etc()	Acquires a semaphore, with extras
create_sem()	Creates a new semaphore
delete_sem()	Deletes a semaphore
get_next_sem_info()	Enumerates information about all semaphores in a given team
get_sem_count()	Returns the value of a semaphore's thread count variable
get_sem_info()	Returns information about a given semaphore
release_sem()	Releases a semaphore, incrementing the thread count by one
release_sem_etc()	Releases a semaphore, incrementing the thread count by a given value
set_sem_owner()	Transfers ownership of a semaphore to a given team

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Semaphore Concepts

A semaphore acts as a key that a thread must acquire in order to continue execution. Any thread that can identify a particular semaphore can attempt to acquire it by passing its `sem_id` identifier—a system-wide number that's assigned when the semaphore is created—to the `acquire_sem()` function. The function blocks until the semaphore is actually acquired.

An alternate function, `acquire_sem_etc()` lets you specify the amount of time you're willing to wait for the semaphore to be acquired, and let you acquire the semaphore more than once in a single go. Unless otherwise noted, characteristics ascribed to `acquire_sem()` apply to `acquire_sem_etc()` as well.)

When a thread acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other threads. The semaphore remains unavailable until it's passed in a call to the `release_sem()` function.

The code that a semaphore "protects" lies between the calls to `acquire_sem()` and `release_sem()`. The disposition of these functions in your code usually follows this pattern:

```
if (acquire_sem(my_semaphore) == B_NO_ERROR) {
   /* Protected code goes here. */
   release_sem(my_semaphore);
}
```

Keep in mind that...

- The calls to the acquire and release functions needn't be locally balanced (although this is by far the most common use). A semaphore can be acquired within one function and released in another. Acquisition and release of the same semaphore can even be performed by two different threads.
- Checking the value returned by `acquire_sem()` is *extremely* important. If an acquire-blocked thread is unblocked by a signal (a return o f `B_INTERRUPTED`), the thread shouldn't procede to the critical section.

The Thread Queue

Every semaphore has its own *thread queue*: This is a list that identifies the threads that are waiting to acquire the semaphore. A thread that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's thread queue where it sits blocked in the `acquire_sem()` call. Each call to `release_sem()` umblocks the thread at the head of that semaphore's queue, thus allowing the thread to return from its call to `acquire_sem()`.

Semaphores don't discriminate between acquisitive threads—they don't prioritize or otherwise reorder the threads in their queues—the oldest waiting thread is always the next to acquire the semaphore.

The Thread Count

To assess availability, a semaphore looks at its *thread count*. This is a counting variable that's initialized when the semaphore is created. Ostensibly, a thread count's initial value (which is passed as the first argument to `create_sem()`) is the number of threads that can acquire the semaphore at a time. (As we'll see later, this isn't the entire story, but it's good enough for now.) For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1—in other words, only one thread can acquire the semaphore at a time.

An initial thread count of 1 is by far the most common use; a thread count of 0 is also useful. Other counts are much less common.

Calls to `acquire_sem()` and `release_sem()` alter the semaphore's thread count: `acquire_sem()` decrements the count, and `release_sem()` increments it. When you call `acquire_sem()`, the function looks at the thread count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function returns immediately.
- If the count is zero or less, the semaphore is unavailable, and the thread is placed in the semaphore's thread queue.

The initial thread count isn't an inviolable limit on the number of threads that can acquire a given semaphore—it's simply the initial value for the sempahore's thread count variable. For example, if you create a semaphore with an initial thread count of 1 and then immediately call `release_sem()` five times, the semaphore's thread count will increase to 6. Furthermore, although you can't initialize the thread count to less-than-zero, an initial value of zero itself is common—it's an integral part of using semaphores to impose an execution order (as demonstrated later).

Summarizing the description above, there are three significant thread count value ranges:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* `acquire_sem()` calls will return without blocking.
- If the count is 0, there are no queued threads, but the next `acquire_sem()` call will block.
- A negative count (*-n*) means there are *n* threads in the semaphore's thread queue, and the next call to `acquire_sem()` will block.

Although it's possible to retrieve the value of a semaphore's thread count (by looking at a field in the semaphore's `sem_info` structure, as described later), you should only do so for amusement—while you're debugging, for example.

You should never predicate your code on the basis of a semaphore's thread count.

Deleting a Semaphore

Every semaphore is owned by a team (the team of the thread that called **create_sem()**). When the last thread in a team dies, it takes the team's semaphores with it.

Prior to the death of a team, you can explicitly delete a semaphore through the **delete_sem()** call. Note, however, that **delete_sem()** must be called from a thread that's a member of the team that owns the semaphore—you can't delete another team's semaphores.

You're allowed to delete a semaphore even if it still has threads in its queue. However, you usually want to avoid this, so deleting a semaphore may require some thought: When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continue—they all return from **acquire_sem()** at once. You can distinguish between a "normal" acquisition and a "semaphore deleted" acquisition by the value that's returned by **acquire_sem()** (the specific return values are listed in the function descriptions, below).

Inter-application Semaphores

The **sem_id** number that identifies a semaphore is a system-wide token—the **sem_id** values that you create in your application will identify your semaphores in all other applications as well. It's possible, therefore, to broadcast the **sem_id** numbers of the semaphores that you create and so allow other applications to acquire and release them—but it's not a very good idea.

A semaphore is best controlled if it's created, acquired, released, and deleted within the same team.

If you want to provide a protected service or resource to other applications, you should accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Semaphore Examples

The following sections provides examples of typical semaphore use. For the full story on semaphores, see Semaphores.

Semaphore Example 1: Locking

The most typical use of a semaphore is to protect a chunk of code that can only be executed by one thread at a time. The semaphore acts as a lock; `acquire_sem()` locks the code, `release_sem()` releases it. Semaphores that are used as locks are (almost always) created with a thread count of 1.

As a simple example, let's say you keep track of a maximum value like this:

```
/* max_val is a global. */
uint32 max_val = 0;
...

/* bump_max() resets the max value, if necessary. */
void bump_max(uint32 new_value)
{
   if (new_value > max_value)
      max_value = new_value;
}
```

bump_max() isn't thread safe; there's a race condition between the comparison and the assignment. So we protect it with a semaphore:

```
sem_id max_sem;
uint32 max_val = 0;

...
/* Initialize the semaphore during a setup routine. */
status_t init()
{
   if ((max_sem = create_sem(1, "max_sem")) < B_NO_ERROR)
      return B_ERROR;
   ...
}
void bump_max(uint32 new_value)
{
   if (acquire_sem(max_sem) != B_NO_ERROR)
      return;
   if (new_value > max_value)
      max_value = new_value;
   release_sem();
}
```

Semaphore Example 2: Benaphores

A "benaphore" is a combination of an atomic variable and a semaphore that can improve locking efficiency. If you're using a semaphore as shown in the previous example, you should consider using a benaphore instead (if you can).

Here's the example re-written to use a benaphore:

```
sem_id max_sem;
uint32 max_val = 0;
int32 ben_val = 0;

status_t init()
{
   /* This time we initialized the semaphore to 0. */
   if ((max_sem = create_sem(0, "max_sem")) < B_NO_ERROR)
      return B_ERROR;
   ...
}
void bump_max(uint32 new_value)
{
   int32 previous = atomic_add(&ben_val, 1);
   if (previous >= 1)
      if (acquire_sem(max_sem) != B_NO_ERROR)
         goto get_out;

   if (new_value > max_value)
      max_value = new_value;

get_out:
   previous = atomic_add(&ben_val, -1);
   if (previous > 1)
      release_sem(max_sem);
}
```

The point, here, is that **acquire_sem()** is called only if it's known (by checking the previous value of **ben_val**) that some other thread is in the middle of the critical section. On the releasing end, the **release_sem()** is called only if some other thread has since entered the function (and is now blocked in the **acquire_sem()** call). An important point, here, is that the semaphore is initialized to 0.

Semaphore Example 3: Imposing an Execution Order

Semaphores can also be used to coordinate threads that are performing separate operations, but that need to perform these operations in a particular order. In the following example, we have a global buffer that's accessed through separate reading and writing functions. Furthermore, we want writes and reads to alternate, with a write going first.

We can lock the entire buffer with a single semaphore, but to enforce alternation we need two semaphores:

```
sem_id write_sem, read_sem;
char buffer[1024];

/* Initialize the semaphores */
status_t init()
{
   if ((write_sem = create_sem(1, "write")) < B_NO_ERROR) {
      return;
   if ((read_sem = create_sem(0, "read")) < B_NO_ERROR) {
      delete_sem(write_sem);
      return;
   }
}

status_t write_buffer(const char *src)
{
   if (acquire_sem(write_sem) != B_NO_ERROR)
      return B_ERROR;

   strncpy(buffer, src, 1024);

   release_sem(read_sem);
}

status_t read_buffer(char *dest, size_t len)
{
   if (acquire_sem(read_sem) != B_NO_ERROR)
      return B_ERROR;

   strncpy(dest, buffer, len);

   release_sem(write_sem);
}
```

The initial thread counts ensure that the buffer will be written to before it's read: If a reader arrives before a writer, the reader will block until the writer releases the `read_sem` semaphore.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Images

Declared in: be/kernel/image.h
Library: libroot.so
Summary: *more...*

This isn't about graphics. An *image* is compiled code, of which there are three types: app images, library images, and add-on images. An app image is executable code that can be launched. A library image is a collection of shared code that you link against when you're compiling your application. An add-on image is code that an app can load and run while the app itself is running. Note that an add-on can also be an app; in other words, you can create an image that can be launched by itself, or that can be loaded into another application.

For more information on creating and using images, see "Image Concepts".

Image Functions

get_image_info() , get_next_image_info() , image_info

```
status_t get_image_info(image_id image, image_info *info)
```

```
status_t get_next_image_info(team_id team,
    int32 *cookie,
    image_info *info)
```

```
struct {} image_info
```

These functions copy, into the *info* argument, the `image_info` structure for a particular image. The `get_image_info()` function gets the information for the image identified by *image*.

The `get_next_image_info()` function lets you step through the list of a team's images through iterated calls. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns `B_BAD_VALUE` when there are no more images to visit:

```
/* Get the image_info for every image in this team. */
image_info info;
int32 cookie = 0;

while (get_next_image_info(0, &cookie, &info) == B_OK)
   ...
```

The `image_info` structure is:

```
typedef struct {
      image_id id;
       image_type type;
      int32 sequence;
      int32 init_order;
      B_PFV init_routine;
      B_PFV term_routine;
      dev_t device;
      ino_t node;
       char name[MAXPATHLEN];
       void *text;
       void *data;
      int32 text_size;
       int32 data_size;
   } image_info
```

The fields are:

- `id`. The image's **`image_id`** number.
- `type`. A constant (listed below) that tells whether this is an app, library, or add-on image.
- `sequence` and `init_order`. These are zero-based ordinal numbers that give the order in which the image was loaded and initialized, compared to all the other images in this team.
- `init_routine` and `term_routine`. These are pointers to the functions that are used to intialize and terminate the image (more specifically, the image's main thread). The **`B_PFV`** type is a cover for a pointer to a **`(void*)`** function.
- `device`. The device that the image file lives on.
- `node`. The node number of the image file.
- `name`. The full pathname of the file whence sprang the image.
- `text` and `text_size`. The address and the size (in bytes) of the image's text segment.
- `data` and `data_size`. The address and size of the image's data segment.

The self-explanatory **`image_type`** constants are:

`image_type` Constants
`B_APP_IMAGE`
`B_LIBRARY_IMAGE`
`B_ADD_ON_IMAGE`

RETURN CODES

- B_OK. The image was found; *info* contains valid information.
- **`B_BAD_VALUE`**. *image* doesn't identify an existing image, *team* doesn't identify an existing team, or there are no more images to visit.

---

get_image_symbol() , get_nth_image_symbol()

status_t **`get_image_symbol(`**image_id *image*,
char **symbol_name*,
int32 *symbol_type*,
void ***location*`)`

status_t **`get_nth_image_symbol(`**image_id *image*,
int32 *n*,
char **name*,
int32 **name_length*,

int32 *symbol_type*,
void **location**)

`get_image_symbol()` returns, in *location*, a pointer to the address of the symbol that's identified by the *image*, *symbol_name*, and *symbol_type* arguments. An example demonstrating the use of this function is given in "Symbols."

`get_nth_image_symbol()` returns information about the *n*'th symbol in the given image. The information is returned in the arguments:

- *name* is the name of the symbol. You have to allocate the *name* buffer before you pass it in—the function copies the name into the buffer.
- You point *name_length* to an integer that gives the length of the *name* buffer that you're passing in. The function uses this value to truncate the string that it copies into *name*. The function then resets *name_length* to the full (untruncated) length of the symbol's name (plus one byte to accommodate a terminating **`NULL`**). To ensure that you've gotten the symbol's full name, you should compare the in-going value of *name_length* with the value that the function sets it to. If the in-going value is less than the full length, you can then re-invoke `get_nth_image_symbol()` with an adequately lengthened *name* buffer, and an increased *name_length* value.

Keep in mind that *name_length* is reset each time you call `get_nth_image_symbol()`. If you're calling the function iteratively (to retrieve all the symbols in an image), you need to reset the *name_length* value between calls.

- The function sets *symbol_type* to **`B_SYMBOL_TYPE_DATA`** if the symbol is a variable, **`B_SYMBOL_TYPE_TEXT`** if the symbol is a function, or **`B_SYMBOL_TYPE_ANY`** if the executable format doesn't distinguish between the two. The argument's value going into the function is of no consequence.
- The function sets *location* to point to the symbol's address.

To retrieve **`image_id`** numbers on which these functions can act, use the `get_next_image_info()` function. Such numbers are also returned directly when you load an add-on image through the **`load_add_on()`** function.

RETURN CODES

- B_OK. The symbol was found.
- **`B_BAD_IMAGE_ID`**. *image* doesn't identify an existing image.
- **`B_BAD_INDEX`**. *n* is out-of-bounds.

---

load_add_on() , unload_add_on()

`image_id load_add_on(`const char **pathname***`)`**

status_t **`unload_add_on(image_id`** *image***`)`**

`load_add_on()` loads an add-on image, identified by *pathname*, into your application's address space.

- *pathname* can be absolute or relative; if it's relative, it's reckoned off the base path specified by the `ADDON_PATH` environment variable.
- The function returns an **`image_id`** (a positive integer) that represents the loaded image. Image ID numbers are unique across the system.

An example that demonstrates the use of **`load_add_on()`** is given in "Loading an Add-on Image."

You can load the same add-on image twice; each time you load the add-on a new, unique **`image_id`** is created and returned.

`unload_add_on()` removes the add-on image identified by the argument. The image's symbols are removed, and the memory that they represent is freed. If the argument doesn't identify a valid image, the function returns **`B_ERROR`**. Otherwise, it returns **`B_OK`**.

RETURN CODES

Positive **`image_id`** ***`value`*** (load) *or*

- B_OK (unload). Success.
- **`B_ERROR`**. The image couldn't be loaded (for whatever reason), or *image* isn't a valid image ID.

---

load_image()

```
thread_id load_image(int argc,
    const char **argv,
    const char **env)
```

Loads an app image into the system (it *doesn't* load the image into the caller's address space), creates a separate team for the new application, and spawns and returns the ID of the team's main thread. The image is identified by the pathname given in *argv*[0].

The arguments are passed to the image's **`main()`** function (they show up there as the function's similarly named arguments):

- *argc* gives the number of entries that are in the *argv* array.
- The first string in the *argv* array must be the name of the image file. You then install any other arguments you want in the array, and terminate the array with a **`NULL`** entry. Note that the value of *argc* shouldn't count *argv*'s terminating **`NULL`**.
- *envp* is an array of environment variables that are also passed to **`main()`**. Typically, you use the global **`environ`** pointer:

```
extern char **environ;

load_image(..., environ);
```

The *argv* and *envp* arrays are copied into the new thread's address space. If you allocated either of these arrays, it's safe to free them immediately after `load_image()` returns.

The thread that's returned by **`load_image()`** is in a suspended state. To start the thread running, you pass the **`thread_id`** to **`resume_thread()`** or **`wait_for_thread()`**.

An example that demonstrates the use of `load_image()` is given in "Loading an App Image."

RETURN CODES

- *Positive integers*. Success.
- **`B_ERROR`**. Failure, for whatever reason.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Images Function Summary

Function Summary

Function	Description
get_image_info()	Returns information about an image
get_image_symbol()	Returns information about a given symbol in a given image
get_next_image_info()	Enumerates information about all images in a team
get_nth_image_symbol()	Returns information about a given symbol in a given image
load_add_on()	Loads an add-on image into memory
load_image()	Loads an application in a new team
unload_add_on()	Removes an add-on image from memory

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Image Concepts

An *image* is compiled code. There are three types of image:

- An *app image* is an application. Every application has a single app image.
- A *library image* is a dynamically linked library (a "shared library"). Most applications link against the system libraries (**libroot.so**, **libbe.so**, and so on) that Be provides.
- An *add-on image* is an image that you load into your application as it's running. Symbols from the add-on image are linked and references are resolved when the image is loaded. An add-on image provides a sort of "heightened dynamic linking" beyond that of a DLL.

The following sections explain how to load and run an app image, how to create a shared library, and how to create and load an add-on image.

Loading an App Image

Loading an app image is like running a "sub-program." The image that you load is launched in much the same way as had you double-clicked it in the Tracker, or launched it from the command line. It runs in its own team—it doesn't share the address space of the application from which it was launched—and, generally, leads its own life.

Any application can be loaded as an app image; you don't need to issue special compile instructions or otherwise manipulate the binary. The one requirement of an app image is that it must have a **`main()`** function.

To load an app image, you call the **`load_image()`** function:

```
thread_id load_image(int32 argc,
        const char **argv,
        const char **env)
```

The function's first two arguments identify the app image (file) that you want to launch—we'll return to this in a moment. Having located the file, the function creates a new team, spawns a main thread in that team, and returns the **`thread_id`** of the thread to you. The thread isn't running: To make it run you pass the **`thread_id`** to **`resume_thread()`** or **`wait_for_thread()`** (as explained in "Threads and Teams").

The *argc*/*argv* argument pair is copied and forwarded to the new thread's `main()` function:

- The first string in the *argv* array must be the name of the image file that you want to launch; **`load_image()`** uses this string to find the file. You then install any other arguments you want in the array, and terminate the array with a **`NULL`** entry. *argc* is set to the number of entries in the *argv* array (not counting the terminating **`NULL`**). It's the caller's responsibility to free the *argv* array after **`load_image()`** returns (remember—the array is copied before it's passed to the new thread).

- *envp* is an array of environment variables that are also passed to `main()`. Typically, you use the global `environ` pointer (which you must declare as an `extern`—see the example, below). You can, of course, create your own environment variable array: As with the *argv* array, the *envp* array should be terminated with a **`NULL`** entry, and you must free the array when **`load_image()`** returns (that is, if you allocated it yourself—don't try to free `environ`).

The following example demonstrates a typical use of **`load_image()`**. First, we include the appropriate files and declare the necessary variables:

```
#include <image.h> /* load_executable() */
#include <OS.h> /* wait_for_thread() */
#include <stdlib.h> /* malloc() */

char **arg_v; /* choose a name that doesn't collide with argv */
int32 arg_c; /* same here vis a vis argc */
thread_id exec_thread;
int32 return_value;
```

Install, in the `arg_v` array, the "command line" arguments. Let's pretend we're launching a program found in **/boot/home/apps/adder** that takes two integers, adds them together, and returns the result as `main()`'s exit code. Thus, there are three arguments: The name of the program, and the values of the two addends converted to strings. Since there are three arguments, we allocate `arg_v` to hold four pointers (to accommodate the final **`NULL`**). Then we allocate and copy the arguments.

```
arg_c = 3;
arg_v = (char **)malloc(sizeof(char *) * (arg_c + 1));

arg_v[0] = strdup("/boot/home/apps/adder");
arg_v[1] = strdup("5");
arg_v[2] = strdup("3");
arg_v[3] = NULL;
```

Now that everything is properly set up, we call **`load_image()`**. After the function returns, it's safe to free the allocated **`arg_v`** array:

```
exec_thread = load_image(arg_c, arg_v, environ);

while (--arg_c >= 0)
   free(arg_v[arg_c]);

free(arg_v);
```

At this point, `exec_thread` is suspended (the natural state of a newly-spawned thread). In order to retrieve its return value, we use **`wait_for_thread()`** to tell the thread to run:

```
wait_for_thread(exec_thread, &return_value);
```

After `wait_for_thread()` returns, the value of `return_value` should be 8 (i.e. 5 + 3).

Creating a Shared Library

The primary documentation for creating a shared library is provided by MetroWerks in their CodeWarrior manual. Beyond the information that you find there, you should be aware of the following amendments and caveats:

- You mustn't export your library's symbols through the `-export all` linker flag. Instead, use the `__declspec()` directive to export each symbol. The macro is described below. If you're compiling for the PPC, you must also export `#pragma` symbols; to do this from the BeIDE, go to the **Linker/PEF** portion of the **Settings** window and set "Export Symbols" to "Use #pragma".
- The loader looks for libraries by following the `LIBRARY_PATH` environment variable. The default library path looks like this:

```
$ echo $LIBRARY_PATH
%A/lib:/boot/home/config/lib:/boot/beos/system/lib
```

where "%A" means the directory that contains the app that the user is lauching.

Exporting and Importing Symbols

If you're developing a shared library you should explicitly export every global symbol in your library by using the **`__declspec()`** macro. To export a symbol, you declare it thus...

```
__declspec(dllexport) typename
```

...where "_declspec(dllexport)" is literal, and *type* and *name* declare the symbol. Some examples:

```
__declspec(dllexport) char *some_name;
__declspec(dllexport) void some_func() {...}
class __declspec(dllexport) MyView {...}
```

To import these symbols, an app that wants to use your library must "reverse" the declaration by replacing **`dllexport`** with **`dllimport`**:

```
__declspec(dllimport) char *some_name;
__declspec(dllimport) void some_func();
class __declspec(dllimport) MyView;
```

The trouble with this system is that it implies two sets of headers, one for exporting (for building your library) and another for importing (that the library client would use). The Be libraries use macros, defined in be/**BeBuild.h**, that throw the import/export switch so the header files can be unified. For example, here's the macro for **libbe**:

```
#if _BUILDING_be
#define _IMPEXP_BE   __declspec(dllexport)
#else
#define _IMPEXP_BE __declspec(dllimport)
#endif
```

When **libbe** is being built, a private compiler directive defines `_BUILDING_be` to be non-zero, and `_IMPEXP_BE` exports symbols. When a developer includes **BeBuild.h,** the `_BUILDING_be` variable is set to zero, so `_IMPEXP_BE` is set to import symbols.

You may want to emulate this system by defining macros for your own libraries. This implies that you have to define a compiler switch (analogous to **_BUILDING_be**) yourself. Set the switch to non-zero when you're building your library, and then set it to zero when you publish your headers for use by library clients.

Creating and Using an Add-on Image

An add-on image is indistinguishable from a shared library image. Creating an add-on is exactly like creating a shared library, a topic that we breezed through above, but with a couple of minor tweaks:

- The loader looks for add-ons by following the paths in the `ADDON_PATH` environment variable. The default `ADDON_PATH` looks like this:

```
$ echo $ADDON_PATH
%A/add-ons:/boot/home/config/add-ons:/boot/beos/system/add-ons
```

- You have to export your add-on symbols, *and* you also must **`extern`** "C" them. This ensures that the symbol names won't be mangled by the compiler.

Exporting Add-on Symbols

To export your add-on's symbols, declare them thus:

```
extern "C" __declspec(dllexport) void some_func();
extern "C" __declspec(dllexport) int32 some_global_data;
```

To extern a C++ class takes more work. You can't extern the class directly; typically what you do is create (and extern) a C function that covers the class constructor:

```
extern "C" __declspec(dllexport) MyClass *instantiate_my_class();
```

`instantiate_my_class()` is implemented to call the MyClass constructor:

```
MyClass *instantiate_my_class()
{
return new MyClass();
}
```

Loading an Add-on Image

To load an add-on into your application, you call the `load_add_on()` function. The function takes a pathname (absolute or relative to the current working directory) to the add-on file, and returns an **`image_id`** number that uniquely identifies the image across the entire system.

For example, let's say you've created an add-on image that's stored in the file **/boot/home/add-ons/adder**. The code that loads the add-on would look like this:

```
/* For brevity, we won't check errors. */
image_id addon_image;

/* Load the add-on. */
addon_image = load_add_on("/boot/home/add-ons/adder");
```

Unlike loading an executable, loading an add-on doesn't create a separate team, nor does it spawn another thread. The whole point of loading an add-on is to bring the image into your application's address space so you can call the functions and fiddle with the variables that the add-on defines.

Symbols

After you've loaded an add-on into your application, you'll want to examine the symbols (variables and functions) that it has brought with it. To get information about a symbol, you call the `get_image_symbol()` function:

```
status_t get_image_symbol(image_id image,
         char *symbol_name,
         int32 symbol_type,
         void **location)
```

The function's first three arguments identify the symbol that you want to get:

- The first argument is the **`image_id`** of the add-on that owns the symbol.
- The second argument is the symbol's name. This assumes, of course, that you know the name, and that the add-on has declared the name as **`extern`**. In general, using an add-on implies just this sort of cooperation.
- The third argument is a constant that gives the symbol's *symbol type*. There are three types, as given below. If the executable format doesn't distinguish between text and data symbols, then you can use any of these three types—they'll all be the same. If the format *does* distinguish between text and data, then you can either ask for the specific type, or you can ask for **`B_SYMBOL_TYPE_ANY`**.

Constant	Meaning
`B_SYMBOL_TYPE_DATA`	Global data (variables)

B_SYMBOL_TYPE_TEXT	Functions
B_SYMBOL_TYPE_ANY	The symbol lives anywhere

The function returns, by reference in its final argument, a pointer to the symbol's address. For example, let's say the **adder** add-on code looks like this:

```
extern "C" int32 a1 = 0;
extern "C" int32 a2 = 0;
extern "C" int32 adder(void);

int32 adder(void)
{
   return (a1 + a2);
}
```

To examine the variables (a1 and a2), you would call **get_image_symbol()** thus:

```
int32 *var_a1, *var_a2;

get_image_symbol(addon_image, "a1", B_SYMBOL_TYPE_DATA, &var_a1);
get_image_symbol(addon_image, "a2", B_SYMBOL_TYPE_DATA, &var_a2);
```

Here we get the symbol for the adder() function:

```
int32 (*func_add)();
get_image_symbol(addon_image, "adder", B_SYMBOL_TYPE_TEXT, &func_add);
```

Now that we've retrieved all the symbols, we can set the values of the two addends and call the function:

```
*var_a1 = 5;
*var_a2 = 3;
int32 return_value = (*func_add)();
```

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaTheme

Derived from: none
Declared in: be/media/MediaTheme.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A BMediaTheme is an object that, given a BParameterWeb, can create a BView containing all the controls needed to configure the BControllable node represented by the BParameterWeb. You can then add the returned BView to a window so the user can configure the node to their liking.

The resulting BView contains not only the controls for configuring the parameters available in the node, but also indicates the data flow path through the node.

The BeOS includes the standard theme, which provides the system default look to media controls, as seen in the Audio and Video preference applications beginning with Release 4.

If you want to change the look of media controls within the BeOS, you can write your own BMediaTheme and install it as the default system theme by calling `SetPreferredTheme()`. If you want to write a new theme and provide the ability for the user to enable it as the new system theme, you'll need to write an application that calls `SetPreferredTheme()`.

If you want to return to the default system theme, you should call `SetPreferredTheme(NULL)`. Make sure you don't have any theme views instantiated in your application when you do so.

Installing a Theme Add-on

Theme add-ons belong at **/boot/home/config/add-ons/media/themes**. They can have any name you like.

At this time, theme add-on support has not been sufficiently tested and may or may not work. If you have problems with it, please file a bug report at http://bebugs.be.com.

Hook Functions

- `BackgroundBitmapFor()`
- `BackgroundColorFor()`
- `ForegroundColorFor()`
- `MakeControlFor()`
- `MakeViewFor()`

Constructor and Destructor

BMediaTheme()

protected:

```
BMediaTheme(const char *name,|
    const char *info,
    const entry_ref *addOn = NULL,
    int32 themeID = 0)
```

The BMediaTheme constructor initializes the theme so it can be used by the system or by an application. The user-readable theme name is passed as the *name* argument, and some longer description of the theme in *info*.

The *name* and *info* strings are copied; you can dispose of the original strings if you wish. Both strings should be human-readable, as they may be presented to the user in a user interface for selecting a theme to be used.

If the theme lives in an add-on on disk, specify the entry_ref of the add-on file in *addOn* and the ID assigned to the theme by its add-on in *themeID*.

~BMediaTheme()

```
virtual ~BMediaTheme()
```

Disposes of memory used by the theme. If you create a theme, be sure the destructor frees any memory allocated by the theme.

Member Functions

BackgroundBitmapFor()

virtual BBitmap ***BackgroundBitmapFor(**bg_kind *backgroundKind***)**

Returns the BBitmap that should be used as the background for the specified type of user interface object. Possible values for *backgroundKind* are:

Constant	Description
B_GENERAL_BG	Used when nothing else is applicable.
B_SETTINGS_BG	Used for panels or windows full of controls.
B_PRESENTATION_BG	Used for windows that present actual media content.
B_EDIT_BG	Used for windows that edit media content.
B_CONTROL_BG	Used for drawing controls.
B_HILITE_BG	Used when highlighting information, such as selected media data.

BackgroundColorFor()

virtual rgb_color **BackgroundColorFor(**bg_kind *backgroundKind***)**

Returns the rgb_color that should be used as the background for the specified type of user interface object. Possible values for *backgroundKind* are:

Constant	Description
B_GENERAL_BG	Used when nothing else is applicable.
B_SETTINGS_BG	Used for panels or windows full of controls.
B_PRESENTATION_BG	Used for windows that present actual media content.
B_EDIT_BG	Used for windows that edit media content.
B_CONTROL_BG	Used for drawing controls.
B_HILITE_BG	Used when highlighting information, such as selected media data.

ForegroundColorFor()

virtual rgb_color **ForegroundColorFor(**bg_kind *foregroundKind***)**

Returns the rgb_color that should be used as the foreground for the specified type of user interface object. Possible values for *foregroundKind* are:

Constant	Description

B_GENERAL_FG	Used when nothing else is applicable.
B_SETTINGS_FG	Used for panels or windows full of controls.
B_PRESENTATION_FG	Used for windows that present actual media content.
B_EDIT_FG	Used for windows that edit media content.
B_CONTROL_FG	Used for drawing controls.
B_HILITE_FG	Used when highlighting information, such as selected media data.

GetRef()

bool **GetRef(**entry_ref **outRef***)**

Sets *outRef* to be a reference to the add-on file from which the theme was loaded and returns **true**. If the theme wasn't loaded froman add-on, this function returns **false**.

ID()

int32 **ID(**void**)**

Returns the theme's ID, as assigned by the add-on in which the theme resides. The ID is unique only within that add-on.

If the theme isn't in an add-on, 0 is returned.

Info()

const char ***Info(**void**)**

Returns the user-readable long description of the theme, as specified when the theme was created.

MakeControlFor()

virtual BControl ***MakeControlFor(**BParameter **parameter***)** = 0

If you create your own theme, your implementation of this function should return an appropriate BControl to operate on the specified BParameter.

Applications that want to use the theme to create the controls, but handle their layout themselves, can call this function for each parameter in the web, rather than rely upon **ViewFor()**. However, if an application uses `MakeControlFor()` to create individual controls rather than letting **ViewFor()** set up the entire view, the application assumes responsibility for setting the control's value in response to value change messages.

If you want the control to send a specific message when invoked, you should call **`BControl::SetMessage()`** on it.

MakeFallbackViewFor()

```
static BControl *MakeFallbackViewFor(BParameter *control)
```

If you're implementing your own theme, and for whatever reason you want a BParameter to use the standard system theme's BControl, call `MakeFallbackViewFor()` to obtain that BControl.

This can be used if your theme is only intended to augment certain controls, or if you receive a BParameter you don't know anything about.

Your **`MakeControlFor()`** implementation should always call `MakeFallbackViewFor()` if it's asked to create a BControl for a BParameter it doesn't understand. This way, all BParameters will get some form of user interface, even if your theme doesn't specifically know how to handle some of them. This will let your theme remain compatible with future versions of the BeOS.

MakeViewFor()

```
virtual BView *MakeViewFor(BParameterWeb *web,
    const BRect *hintRect = NULL) = 0
```

Given the specified BParameterWeb and *hintRect*, your implementation of this function should construct a BView that contains the BControls for manipulating the BParameters in the *web*. The *hintRect* is an area the caller thinks is appropriate for you to fill with your BView; your theme should try to stay within that rectangle if possible.

The *web* returned by this function belongs to the theme; applications shouldn't delete it (a properly-written theme will automatically dispose of the *web* when the view is closed).

This function is called by **`ViewFor()`**.

PreferredTheme() , SetPreferredTheme()

```
static BMediaTheme *PreferredTheme(void)
static status_t *SetPreferredTheme(BMediaTheme *newPreferredTheme)
```

`PreferredTheme()` returns the current preferred theme; `SetPreferredTheme()` changes the preferred theme to the theme specified by *newPreferredTheme*.

The preferred theme is the theme that will be used for creating node controlling user interfaces if no specific theme is requested when the interface is created.

RETURN CODES

- B_OK. The preferred theme was changed without error.
- **B_NOT_ALLOWED**. Access to the settings file denied.
- Other errors. Directory and file I/O errors may occur while updating the settings file.

See also: **ViewFor()**, **MakeViewFor()**

Name()

```
const char *Name(void)
```

Returns the user-readable name of the theme, as specified when the theme was created.

Set see PreferredTheme()

ViewFor()

```
static BView *ViewFor(BParameterWeb *web,
    const BRect *hintRect = NULL,
    BMediaTheme *usingTheme = NULL)
```

Given a BParameterWeb *web*, typically returned by **BMediaRoster::GetControlWeb()**, this function creates a BView suitable for adding to a BWindow to allow the user to configure the node the web describes, including a diagram indicating the data flow path through the node, and controls to let the user configure each control point.

This function is the public interface for creating a view for configuring a node (the **MakeViewFor()** function is the hook you override if you're implementing your own theme).

The returned view is created using the theme specified by *usingTheme*; if this argument is **NULL**, the preferred theme is used. The *hintRect* parameter specifies the rectangle the view should try to occupy, and is passed to **MakeViewFor()**.

Global C Functions

get_nth_theme()

```
status_t get_theme_at(int32 n, const char **outName,
    const char **outInfo, int32 *outID)
```

This function is called after the theme add-on is loaded, to determine what themes are available in the add-on. It's called repeatedly, with successively-higher values of *n*, until it returns an error.

Your add-on's implementation of this function should set *outName* to point to the *n*th theme's name, *outInfo* to point to information describing the *n*th theme, and *outID* should be set to the *n*th theme's ID number, which used internally by your add-on to distinguish among the themes it supports.

make_theme()

```
BMediaTheme *make_theme(int32 themeID, image_id imageID)
```

If you're writing a theme to be loaded from an add-on file, you must implement `make_theme()` to create a BMediaTheme for the specified *themeID*. *imageID* is the image_id of the add-on in which the theme is located.

Constants

Background Color and Bitmap Kinds

Constant	Description
B_GENERAL_BG	Used when nothing else is applicable.
B_SETTINGS_BG	Used for panels or windows full of controls.
B_PRESENTATION_BG	Used for windows that present actual media content.
B_EDIT_BG	Used for windows that edit media content.
B_CONTROL_BG	Used for drawing controls.
B_HILITE_BG	Used when highlighting information, such as selected media data.

These constants identify parts of the user interface that a theme can customize the appearance of by altering the color or bitmap that's used in the background for various user interface elements. They're used when calling `BackgroundBitmapFor()` and `BackgroundColorFor()`.

For example, the `B_PRESENTATION_BG` color and/or bitmap would be used as the background of a window in which an oscilloscope display shows the waveform of a sound wave being played back, and a window used for editing the waveform would have its' background in the color specified by `B_EDIT_BG` (or using the bitmap specified by `B_EDIT_BG`, if one exists), except for the part of the wave that's selected by the user, which would have the background color or bitmap specified by `B_HILITE_BG`.

Foreground Color Kinds

Constant	Description
B_GENERAL_FG	Used when nothing else is applicable.
B_SETTINGS_FG	Used for panels or windows full of controls.
B_PRESENTATION_FG	Used for windows that present actual media content.
B_EDIT_FG	Used for windows that edit media content.
B_CONTROL_FG	Used for drawing controls.
B_HILITE_FG	Used when highlighting information, such as selected media data.

The foreground color kinds are used when calling `ForegroundColorFor()` to obtain the color to use when drawing the foreground portions of various interface elements.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaTheme Function Summary

Function Summary

Function	Description
BackgroundBitmapFor()	Returns the BBitmap that should be used as the background for the specified type of user interface object
BackgroundColorFor()	Returns the color that should be used as the background for the specified type of user interface object
ForegroundColorFor()	Returns the color that should be used as the foreground for the specified type of user interface object
GetRef()	Returns the entry_ref of the add-on from which the theme was loaded
get_nth_theme()	Hook: return information about a theme in an add-on file
ID()	Returns the theme's ID
Info()	Returns the theme's long description
MakeControlFor()	Hook: create a BControl to implement a specified BParameter
MakeFallbackViewFor()	Creates a BControl in the system theme
MakeViewFor()	Hook: return a BView containing

	BControls for each BParameter in the web
make_theme()	Hook: create and return a BMediaTheme object representing a theme loaded from disk
BMediaTheme()	Constructor
~BMediaTheme()	Destructor
Name()	Returns the theme's name
PreferredTheme()	Returns the preferred theme
SetPreferredTheme()	Sets the preferred theme
ViewFor()	Returns a BView that can be presented to configure the BControllable node

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameterWeb

Derived from: BFlattenable
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

BParameterWeb serves as a container for information describing the relationships among the configurable parameters of a BControllable node. A BControllable subclass should, when constructed, create a BParameterWeb and populate it with controls through one or more BParameterGroups. This web of parameters describes the signal path within the node, and the points at which the node's manipulation of the data can be controlled.

The Audio control panel is actually derived from the BParameterWeb of the node currently selected as the default audio input node.

The BParameterWeb lets client applications query a node to determine how it can be configured, so the application can then create and display a user interface permitting the user to configure the node using standard user interface objects (like sliders, checkboxes, lines and arrows indicating the data path, and so forth).

By using the parameter web, device-independence is maintained without sacrificing the ability to create generic code to configure devices.

Constructor and Destructor

BParameterWeb()

```
BParameterWeb()
```

The BParameterWeb constructor. You'll usually create one BParameterWeb object per BControllable node; to attach a BParameterWeb to a BControllable node, you should call **`BControllable::SetParameterWeb()`**.

~BParameterWeb()

```
~BParameterWeb()
```

Once you've called **`BControllable::SetParameterWeb()`**, the node takes responsibility for the parameter web object and you shouldn't delete it.

If you don't call **`BControllable::SetParameterWeb()`**, then delete the BParameterWeb object when you're done with it.

Member Functions

CountGroups()

int32 **CountGroups**(void)

Returns the number of BParameterGroup objects that are currently attached to the BParameterWeb.

CountParameters()

int32 **CountParameters**(void)

Returns the number of BParameters in the entire web, including those in all BParameterGroups attached to it.

GroupAt()

BParameterGroup ***GroupAt**(int32 *index*)

Returns the BParameterGroup located at the specified index within the list of groups contained by the BParameterWeb.

The first group is numbered 0, so the maximum legal value for *index* is **CountGroups()**-1. If the specified *index* is outside that range, **NULL** is returned.

MakeGroup()

BParameterGroup ***MakeGroup**(const char **name*)

Creates a new BParameterGroup object to be used for grouping parameters within the BParameterWeb, and attaches it to the BParameterWeb.

All BParameters created in the group will belong to that group, and, by recursion, to the web itself.

You can nest BParameterGroups if you want to; however, BParameters can't be shared among multiple groups.

Node()

media_node **Node**(void)

Returns the media_node for the BControllable node that owns this BParameterWeb object.

If the BParameterWeb hasn't been attached to a BControllable node yet, **media_node::null** is returned.

ParameterAt()

BParameter ***ParameterAt**(int32 *index*)

Returns the BParameter at the specified *index* within the entire BParameterWeb, including those in all attached groups. The first parameter is numbered 0, so the maximum legal value for *index* is **CountParameters()**-1. If the specified *index* is outside that range, **NULL** is returned.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFlattenable

Derived from: none
Declared in: be/support/Flattenable.h
Library: libbe.so
Summary: *more...*

The BFlattenable class declares a protocol for objects that can be flattened—written to an untyped buffer of bytes—and unflattened from the buffer. By implementing this protocol, a class gives others the ability to manipulate its objects in a flexible manner. Currently, only the BMessage class, through its `AddFlat()` and `FindFlat()` functions, declares an API that can deal with BFlattenable objects. The BPath class in the Storage Kit derives from BFlattenable.

As this class merely declares a protocol for other classes to implement, it doesn't include a constructor or destructor.

Member Functions

Allows see TypeCode()

Flatten() , Unflatten()

virtual status_t **Flatten**(void *_buffer_, ssize_t _numBytes_) const = 0
virtual status_t **Unflatten**(type_code _code_, const void *_buffer_, ssize_t _numBytes_) = 0

`Flatten()` is implemented by derived classes to write the object into the *buffer*. There are *numBytes* bytes of memory available at the *buffer* address. If this isn't at least as much memory as the `FlattenedSize()` function says is necessary, `Flatten()` should return an error. If successful, it should return `B_OK`.

`Unflatten()` is implemented by derived classes to set object values from *numBytes* bytes of data taken from the *buffer*. However, it should read the data only if the type *code* it's passed indicates that the data is a type that it supports—that is, only if its `AllowsTypeCode()` function returns `true` for the *code*. If successful in reconstructing the object from the flattened data, `Unflatten()` should return `B_OK`. If not, it should return `B_ERROR` or a more descriptive error code.

FlattenedSize()

virtual ssize_t **FlattenedSize**(void) const = 0

Implemented by derived classes to return the amount of memory needed to hold the flattened object. This is the minimal amount that must be allocated and passed to `Flatten()`.

IsFixedSize()

virtual bool **IsFixedSize**(void) const = 0

Implemented by derived classes to return **`true`** if all instances of the class take up the same amount of memory when they're flattened, and **`false`** if their flattened sizes can differ. The sizes will differ, for example, if a variable-length string is part of the flattened data.

TypeCode() , AllowsTypeCode()

virtual type_code **TypeCode**(void) const = 0
virtual bool **AllowsTypeCode**(type_code _code_) const

`TypeCode()` is implemented by derived classes to return the type code that identifies the class type. The code is used to identify an instance of the class in its flattened state, for example when it's added to a BMessage.

`AllowsType()` returns `true` if the *code* it's passed matches the code returned by `TypeCode()` and `false` if not. If can be modified in derived classes to apply a more liberal standard—to allow more than one type code to identify the object.

See also: `BMessage::AddData()`

Unflatten() see Flatten()

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFlattenable Function Summary

Function Summary

Function	Description
AllowsTypeCode()	Implemented to test if type code matches type code for the class
Flatten()	Implemented to write object to an untyped buffer
FlattenedSize()	Implemented to return size of flattened object
IsFixedSize()	Implemented to test if flattened objects are all the same size
TypeCode()	Implemented to return the identifying type code
Unflatten()	Implemented to read object from an untyped buffer

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Application Kit

BApplication

Declared in: be/app/Appplication.h
more...

SetCursor()

```
void SetCursor(const BCursor *cursor, bool sync = true)
```

The new **SetCursor()** function sets the cursor as a BCursor object. If *sync* is **true**, the function doesn't return until the cursor has actually been changed; otherwise, the function returns immediately.

BCursor

Declared in: be/app/Cursor.hp/headers/be/app/Cursor.h

The new BCursor class and related API let you retrieve and set the cursor on an app-wide and view-specific basis. The two Be-defined cursors, which used to be identified by name, are now represented by global BCursor objects (from **app/AppDefs.h**):

- **`B_CURSOR_SYSTEM_DEFAULT`** is the "hand" cursor.
- **`B_CURSOR_I_BEAM`** is the insertion point cursor commonly used by text editors.

The functions that control the app-wide cursor (**`SetCursor()`**, **`ShowCursor()`**, etc) are, as before, defined by BApplication; the only protocol difference is that **`SetCursor()`** now takes a BCursor object as an argument.

To set the cursor for a specific BView, use the new **`BView::SetViewCursor()`** function. In addition to an initial BCursor* argument, the function takes an optional boolean argument that, when **`true`**, tells the function to call **`Sync()`** before returning. This *sync* argument is **`true`** by default. With this new API, the BView doesn't need to watch for mouse-entered and mouse-exited events; the cursor will change automatically as the mouse enters and exits the view.

The old **`B_HAND_CURSOR`** and **`B_I_BEAM_CURSOR`** constants (app/AppDefs.h) have been deprecated.

BHandler

Declared in: be/app/Handler.h

BHandler has a new "observe-notify" system in which a handler (the "observer") can register itself to be sent notifications from some other handler (the "notifier"). To register as an observer, a BHandler calls `StartWatching()`. When the notifier wants to broadcast a message to its observers, it calls `SendNotices()`.

The Beta 1 versions of these functions will change to match the protocol given here.

StartWatching() , StartWatchingAll() , StopWatching() , StopWatchingAll()

status_t StartWatching(BMessenger *notifier*, **uint32** *what***)**
status_t StartWatchingAll(BMessenger *notifier***)**
status_t StartWatching(BHandler **notifier*, **uint32** *what***)**
status_t StartWatchingAll(BHandler **notifier***)**
status_t StopWatching(BMessenger *notifier*, **uint32** *what***)**
status_t StopWatchingAll(BMessenger *notifier***)**
status_t StopWatching(BHandler **notifier*, **uint32** *what***)**
status_t StopWatchingAll(BHandler **notifier***)**

StartWatching[All]() places this BHandler on *notifier*'s list of observers. When the notifier broadcasts a notification, this object will receive a `B_OBSERVER_NOTICE_CHANGE` message.

The `StartWatching()` *what* argument lets the observer tell the notifier which sort of changes it wants to be notified about (where the values that *what* can take are defined by the notifier). A handler can register for multiple *what* notifications with the same notifier by calling `StartWatching()` multiple times. For example:

```
/* It's assumed that notifier has defined the NEW_... values */
handler->StartWatching(notifier, NEW_NAME);
handler->StartWatching(notifier, NEW_LOCATION);
handler->StartWatching(notifier, NEW_ATTITUDE);
```

To receive notifications of all changes (as defined by *notifier*), use the `StartWatchingAll()` functions.

The *notifier*-as-BMessenger version of `StartWatching()` lets you watch a notifier that's local or remote. The BHandler* version can only watch local notifiers, but it can be called it before *notifier* is attached to a BLooper.

StopWatching() unregisters an observer for a given *notifier*/*what* combination. StopWatchingAll() completely unregisters the observer for that notifier.

The *what* value(s) that an observer registers for here shows up in the notification message as the **B_OBSERVE_WHICH_CHANGE** field. The notification message itself has a **what** of B_OBSERVER_NOTICE_CHANGE. For example, the handler that registered for the "NEW_*" changes above would respond to notifications thus:

```
void myHandler::MessageReceived(BMessage *msg)
{
   uint32 change;
   switch (msg->what) {
   case B_OBSERVER_NOTICE_CHANGE:
      msg->FindField(B_OBSERVE_WHICH_CHANGE, &change);
      switch (change) {
      case NEW_NAME:
         ...
      case NEW_LOCATION:
         ...
      case NEW_ATTITUDE:
         ...
      ...
      }
   ...
   }
   ...
}
```

Keep in mind that the notifications are broadcast at the discretion of the notifier, through its SendNotices() function.

RETURN CODES

- B_OK. Success.
- B_BAD_PORT_ID. *notifier* (as a BMessenger) is invalid.
- *Other errors*. ...

SendNotices()

```
virtual void SendNotices(uint32 what, const BMessage *msg = 0)
```

Sends a B_OBSERVER_NOTICE_CHANGE message to each BHandler object (or "observer") that's observing this handler (the "notifier"). To observe a notifier, the observer calls StartWatching(). The *what* argument describes the type of change that's prompting this notification; only those observers that have registered to be notified about *what* (or that are watching all changes) are sent notifications.

The **B_OBSERVER_NOTICE_CHANGE** messages that are sent are copied from *msg* with the *what* argument added as the "be:old_what" field. Note that *msg*'s original **what** field is clobbered.

BMessage

Declared in: be/app/Message.h

BMessage()

The BMessage constructor that takes a (BMessage *) argument has been retired:

BMessage(**BMessage** **msg*); *obsolete—do not use*

To replace the constructor in your code, simply de-reference the argument. For example:

```
/* OLD CODE: */
void SomeFunc(BMessage *msg)
{
   BMessage new_msg(msg);
   ...
}

/* NEW CODE: */
void SomeFunc(BMessage *msg)
{
   BMessage new_msg(*msg);
   ...
}
```

This will cause *msg* to be copied into *new_msg*, but (to anticipate your objection) it *won't* incur a second (leaked) copy.

FindInt8() , FindInt64()

int8 FindInt8(const char **name*, **int32** *index* = 0**) const**
int64 FindInt64(const char **name*, **int32** *index* = 0**) const**

New direct-return versions of these functions (previously you could only retrieve an **int8** or an **int64** by reference).

BRoster

Declared in: be/app/Roster.h

BRoster now provides access to the system-wide "recent documents" and "recent applications" lists. Also, some versions of `Launch()` have been deprecated in favor of new heavily **const**'d versions.

GetRecentDocuments() , GetRecentFolders() , GetRecentApps() , AddToRecentDocuments() , AddToFolders()

```
void GetRecentDocuments(BMessage *refList
    int32 maxCount,
    const char *fileType = NULL,
    const char *appSig = NULL) const
void GetRecentDocuments(BMessage *refList
    int32 maxCount,
    const char *fileTypes[],
    int32 fileTypesCount,
    const char *appSig = NULL) const
```

```
void GetRecentFolders(BMessage *refList
    int32 maxCount,
    const char *appSig = NULL) const
```

```
void GetRecentApps(BMessage *refList
    int32 maxCount) const
```

```
void AddToRecentDocuments(const entry_ref *ref
    const char *appSig = NULL) const
```

```
void AddToRecentFolders(const entry_ref *ref
    const char *appSig = NULL) const
```

These functions access the system-wide "recently opened documents," "recently opened folders," and "recently launched apps" lists. These lists are displayed by file panels, **Deskbar**, and other parts of the system. You can add to the document and folder lists, but not the app list (it's updated by the system). You can't programatically delete items from these lists.

The **`Get...`** functions retrieve the items as `entry_ref`s which they add to the "refs" field of the *refList* BMessage argument (*refList* must be allocated by the caller). The items in "refs" are listed from most recent to least recent.

`GetRecentDocuments()` gets the `entry_ref`s for the first *maxCount* items in the recent documents list. The optional *fileType* and *appSig* arguments let you specify that you only want to see files of a certain type and/or that were opened by a particular app. The *fileTypes*/*fileTypesCount* version lets you specify a list of candidate file types.

`GetRecentFolders()` returns the first *maxCount* `entry_ref`s in the recent folders list. The optional *appSig* lets you ask for folders that were opened by a particular app (only).

`GetRecentApps()` returns the first *maxCount* `entry_ref`s in the recent apps list.

`AddToRecentDocuments()` and `AddToRecentFolders()` adds the argument *ref* to the named list. *appSig*, if supplied, is the signature of the application that opened the document or folder.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Release Notes Table of Contents

- The Application Kit
- The Game Kit
- The Interface Kit
- The Kernel Kit
- The Media Kit
- OpenGLKit
- The Storage Kit
- The Support Kit
- Deskbar
- System Messages
- The Translation Kit
- Deprecated API

The Be Book Release Notes Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Be Book Release Notes

Release Notes: Master Index

A

AddItem()	Deskbar
AddToFolders()	The Application Kit
AddToRecentDocuments()	The Application Kit
and	The Kernel Kit
The Application Kit	The Application Kit
The Application Kit	The Application Kit
BApplication	The Application Kit

B

BarThickness()	The Interface Kit
BBufferConsumer	The Media Kit
BBufferConsumer()	The Media Kit
BBufferProducer	The Media Kit
BBufferProducer()	The Media Kit

C

BChannelControl and BChannelSlider	The Interface Kit
Channel Control Message	System Messages
CheckSnifferRule()	The Storage Kit

Constants and Types	The Media Kit
Constants	Deprecated API
ControlThread	The Media Kit
Cursor	Deprecated API
BCursor	The Application Kit

D

Deprecated API	Deprecated API
Deprecated API	Deprecated API
Deskbar	Deskbar
Deskbar	Deskbar
BDeskbar::AddItem()	Deprecated API
does_file_accept_format()	The Media Kit

E

enum	The Media Kit

F

BFileGameSound	The Game Kit
BFileGameSound()	The Game Kit
FindInt64()	The Application Kit
FindInt8()	The Application Kit
FindKeyFrameForFrame()	The Media Kit
FindKeyFrameForTime()	The Media Kit

G

BGLView	OpenGLKit
The Game Kit	The Game Kit
The Game Kit	The Game Kit
GetAVIFormatFor()	The Media Kit
GetAllInputsFor()	The Media Kit
GetAllOutputFor()	The Media Kit
GetAudioOutput()	The Media Kit
GetBeOSFormatFor()	The Media Kit
GetBitmap()	The Translation Kit
GetEncodeParameters()	The Media Kit
GetFormatFor()	The Media Kit
GetInstancesFor()	The Media Kit
GetMode()	The Interface Kit
GetPredicate()	The Storage Kit
GetQuicktimeFormatFor	The Media Kit
GetRecentApps()	The Application Kit
GetRecentDocuments()	The Application Kit
GetRecentFolders()	The Application Kit
GetSnifferRule()	The Storage Kit
GetVolumeInfo()	The Media Kit
get_cpuid(), cpuid_info	The Kernel Kit
get_mouse_acceleration()	The Interface Kit
get_next_encoder()	The Media Kit

Global Functions	The Translation Kit
GuessMimeType()	The Storage Kit

H

BHandler	The Application Kit
hash_mark_location	The Interface Kit
HideTyping()	The Interface Kit

I

Images	The Kernel Kit
Initialization	The Kernel Kit
instantiate_deskbar_item()	Deskbar
The Interface Kit	The Interface Kit
The Interface Kit	The Interface Kit
IsPaused()	The Game Kit
IsTypingHidden()	The Interface Kit

K

The Kernel Kit	The Kernel Kit
The Kernel Kit	The Kernel Kit
Kit	Deprecated API

M

MakeFormatFor()	The Media Kit
make_format_flags	The Media Kit
make_nth_translator()	The Translation Kit
BMediaAddOn	The Media Kit
Media	Deprecated API
BMediaEventLooper	The Media Kit
BMediaFile	The Media Kit
BMediaFile()	The Media Kit
BMediaFormats	The Media Kit
Media Kit Miscellanea	The Media Kit
The Media Kit	The Media Kit
The Media Kit	The Media Kit
BMediaRoster	The Media Kit
BMediaTrack	The Media Kit
media_avr_description	The Media Kit
Memory	The Kernel Kit
BMenu	The Interface Kit
menu_tracking_hook	The Interface Kit
BMessage	The Application Kit
BMessage()	The Application Kit
BMimeType	The Storage Kit
Mouse Acceleration	The Interface Kit
B_MOUSE_WHEEL_CHANGED	System Messages

N

NodeIDFor()	The Media Kit
BNode	The Storage Kit
NotifyFlavorChange()	The Media Kit

O

B_OBSERVER_NOTICE_CHANGE	System Messages
on_exit_thread(), exit_thread_func	The Kernel Kit
OpenGLKit	OpenGLKit
OpenGLKit	OpenGLKit
BOptionControl and BOptionPopUp	The Interface Kit
Orientation()	The Interface Kit

P

BPath	The Storage Kit
BPath()	The Storage Kit
ProposeMode()	The Interface Kit

Q

BQuery	The Storage Kit

R

ReadAttrString()	The Storage Kit
ReadFrames()	The Media Kit

BRegion	The Interface Kit
BRegion()	The Interface Kit
BRoster	The Application Kit
Routines	The Kernel Kit
rtm_realloc	The Media Kit

S

BScreen	The Interface Kit
BScreen::ProposeDisplayMode()	Deprecated API
SendNotices()	The Application Kit
SetAudioOuput()	The Media Kit
SetBarThickness()	The Interface Kit
SetCursor()	The Application Kit
SetMode()	The Interface Kit
SetOrientation()	The Interface Kit
SetPaused()	The Game Kit
SetSnifferRule()	The Storage Kit
SetTo()	The Storage Kit
SetTrackingHook()	The Interface Kit
SetViewCursor()	The Interface Kit
SetVolumeDB()	The Media Kit
set_mouse_acceleration()	The Interface Kit
BSimpleGameSound	The Game Kit
BSimpleGameSound()	The Game Kit
BSlider	The Interface Kit

BSlider	The Interface Kit
B_SNIFFER_RULE_CHANGED	The Storage Kit
BSoundPlayer	The Media Kit
BSoundPlayer()	The Media Kit
StartWatchingAll()	The Application Kit
StartWatching()	The Application Kit
StopWatchingAll()	The Application Kit
StopWatching()	The Application Kit
The Storage Kit	The Storage Kit
The Storage Kit	The Storage Kit
suggestion_quality	The Media Kit
The Support Kit	The Support Kit
The Support Kit	The Support Kit
SwapBuffers()	OpenGLKit
System Beep	The Support Kit
System Messages	System Messages
System Messages	System Messages

T

Termination	The Kernel Kit
BTextView	The Interface Kit
BTimedEventQueue	The Media Kit
The Translation Kit	The Translation Kit
The Translation Kit	The Translation Kit
BTranslationUtils	The Translation Kit

Translator Add-ons	The Translation Kit
typedef	The Media Kit

U

UI Colors	The Interface Kit

V

BView	The Interface Kit
VolumeDB()	The Media Kit

W

WriteAttrString()	The Storage Kit
WriteFrames()	The Media Kit

The Be Book Release Notes

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Deskbar

Declared in: be/be_apps/Deskbar/Deskbar.h
more...

There's a new system for adding items to **Deskbar**'s shelf. Instead of using "live" BView objects, you create an add-on that implements the **Deskbar** item-instantiation function, and then pass a pointer to the add-on to **BDeskbar::AddItem()**.

An advantage of the add-on system is that **Deskbar** items are now persistent across boots.

---

instantiate_deskbar_item()

```
BView *instantiate_deskbar_item(void)
```

In Beta 1, this function is called **instantiate_deskbar_addon()**. The name will be changed in the next Beta.

This is the deskbar-item instantiation function that you implement in an add-on. The function should return a BView object that knows how to describe itself in an archive. The add-on that contains this function doesn't have to live in a particular directory.

AddItem()

```
status_t AddItem(entry_ref *addon, int32 *id = NULL)
status_t AddItem(BView *view, int32 *id = NULL)  deprecated
```

AddItem() puts a new item on the **Deskbar**'s shelf. To create the new item, **Deskbar** loads *addon* and calls the **instantiate_deskbar_item()** function that should be implemented therein. The *id* argument, if supplied, is set to the unique id number that **Deskbar** generates for the new item.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Interface Kit

Mouse Acceleration

Declared in: be/interface/InterfaceDefs.h
more...

---

get_mouse_acceleration() , set_mouse_acceleration()

```
status_t get_mouse_acceleration(int32 *accel)
status_t set_mouse_acceleration(int32 accel)
```

These new functions let you get and set the "non-linear mouse speed scalar." The higher the acceleration factor, the farther the mouse travels as the user increases mouse velocity. An acceleration of 0 means the mouse's velocity isn't taken into consideration—if the user moves the (physical) mouse one inch, the cursor on-screen moves the same distance no matter how fast the user moved the mouse. The default acceleration is about 65000. The upper practical limit depends on who your mouse is, but 1000000 is probably about as high as you ever want to go.

Note that mouse acceleration is independent of mouse speed (see `get_mouse_speed()`). Currently, there's no user UI for setting mouse acceleration, only mouse speed.

UI Colors

Declared in: be/interface/InterfaceDefs.h

```
enum color_which {
    B_PANEL_BACKGROUND_COLOR,
    B_MENU_BACKGROUND_COLOR,
    B_MENU_SELECTION_BACKGROUND_COLOR, new
    B_MENU_ITEM_TEXT_COLOR, new
    B_MENU_SELECTED_ITEM_TEXT_COLOR, new
    B_WINDOW_TAB_COLOR,
    B_KEYBOARD_NAVIGATION_COLOR,
    B_DESKTOP_COLOR
        };
```

The **color_which** constants represent the colors that are used by the Be UI. Of these colors, **B_MENU_SELECTION_BACKGROUND_COLOR**, **B_MENU_ITEM_TEXT_COLOR**, and **B_MENU_SELECTED_ITEM_TEXT_COLOR** are new. Use **ui_color()** to convert a **color_which** to an **rgb_color**.

BChannelControl and BChannelSlider

Declared in: be/interface/ChannelControl.hnterface/ChannelControl.h
Declared in: be/interface/ChannelSlider.h

The new BChannelControl class (which derives from BControl) is an abstract class for UI objects that want to control more than one "channel" of values, such as a stereo volume control, or an rgb slider trio. Each of the channels has its own independent value. BChannelControl values are **`int32`**.

The most important BChannelControl member functions are:

- **`SetChannelCount()`** sets the number of channels that the object will control.
- **`SetValueFor()`** sets the current value for a given channel.
- **`SetLimitsFor()`** sets the minimum and maximum values for a given channel
- **`SetLimitLabelsFor()`** sets the labels that display the minimum and maximum values.
- **`SupportsIndividualLimits()`** must be implemented by every BChannelControl subclass. If the subclass supports independent channel value limits (and labels), then the function should be implemented to return **`true`**. If every channel is restricted to the same value range (and the same limit labels), the function should return false.

When the user manipulates a BChannelControl object, the object adds an array of **`int32`** values (one per channel) to the "be:channel_value" field of the BMessage that's sent to the object's target.

BChannelSlider is an implementation of BChannelControl that draws a (horizontal or vertical) slider for each channel. When the user clicks or drags the control with the first mouse button, all of the channels snap to the same value and drag together. The second mouse button moves the channel sliders independently of each.

BMenu

Declared in: be/interface/Menu.h

BMenu provides a new mechanism whereby you can track the user's movements within your menu—without having to write a BMenu subclass. Instead, you provide a "tracking hook" that's invoked *whenever* the mouse moves while your menu is open.

SetTrackingHook()

```
void SetTrackingHook(menu_tracking_hook *func, void *state)
```

Sets the callback function that's called when the object is tracking the mouse. Any mouse movement will cause *func* to be invoked—it isn't just invoked when the mouses moves to a new menu items (for example). The *state* data is passed as an argument to *func*. Note that *state* is not copied.

---

menu_tracking_hook

```
typedef bool (* menu_tracking_hook )(BMenu *, void *)
```

Protocol for the menu-tracking hook function. The first argument is the BMenu that's being tracked; the second argument is the *state* argument that was passed to **SetTrackingHook()**. If the function returns **true**, the BMenu stops tracking the mouse; if it returns **false**, the tracking continues.

BOptionControl and BOptionPopUp

Declared in: be/interface/OptionControl.hinterface/OptionControl.h
Declared in: be/interface/OptionPopUp.h

The BOptionControl class (which derives from BControl) is an abstract class that associates a set of names with a set of values. The names are displayed to the user; when the user selects a name, the corresponding value is returned in the value-changed message (in the normal "be:value" field). The options in a BOptionControl object are mutually exclusive.

The most important member functions are:

- **`AddOption()`** adds a name/value pair.
- **`SelectOptionFor()`** selects an option given a name or a value.
- **`SelectedOption()`** is an abstract functions that returns (directly) the index of the currently selected option, and returns (by reference) the option's name and value. Each subclass must implement this function.

The BOptionPopUp class is an implementation of BOptionControl that presents the options in a pop up menu.

BRegion

Declared in: be/interface/Menu.h

BRegion()

```
BRegion(const BRect rect)
```

New BRegion constructor creates a region that matches the bounds of the given *rect*.

BScreen

Declared in: be/interface/Screen.h

ProposeMode()

This is the new name for the previously-existing **`ProposeDisplayMode()`** function. The latter name is deprecated.

GetMode() , SetMode()

```
status_t GetMode(uint32 workspace,
    display_mode *mode)
```

```
status_t SetMode(uint32 workspace,
    display_mode *mode,
    bool makeDefault = false)
```

These two functions are workspace-specific versions of the previously-existing **`GetMode()`** and **`SetMode()`** functions. A workspace is indentified by its index (0-based). The old (non-*workspace*) versions work on the current workspace.

BSlider

Declared in: be/interface/Slider.h

Slap your controls into an upright position—BSliders can now slide up and down.

BSlider

```
BSlider(...,orientation posture, ...)
```

A new version of the BSlider constructor takes an additional posture argument that can be B_HORIZONTAL or B_VERTICAL. Sliders are horizontal by default.

---

hash_mark_location

```
enum hash_mark_location {
    ...,
    B_HASH_MARKS_LEFT = 1,
    B_HASH_MARKS_RIGHT = 2,
    ... };
```

These two new values are accepted by the SetHashMarks() function to declare the presence of the right and left hash marks.

Orientation() , SetOrientation()

```
orientation Orientation(void)
virtual void SetOrientation(orientation which)
```

These new functions return and set the slider's orientation as B_HORIZONTAL or B_VERTICAL.

BarThickness() , SetBarThickness()

```
float BarThickness(void)
virtual void SetBarThickness(float thickness)
```

These new functions return and set the thickness, in pixels, of the slider's "bar" (the track that the thumb slides along).

BTextView

Declared in: be/interface/TextView.h

HideTyping() , IsTypingHidden()

void HideTyping(bool *hideIt***)**
bool IsTypingHidden(void) const

HideTyping(true) tells the object to print an asterisk for each keystroke, thereby hiding the actual text typed. Perfect for password panels and other paranoia.

BView

Declared in: be/interface/View.h

SetViewCursor()

```
void SetViewCursor(const BCursor *cursor, bool sync=true)
```

Sets the cursor that's displayed when the mouse enters this views bounds. If *sync* is **true**, the function doesn't return until the cursor has actually been changed; otherwise, the function returns immediately.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Input Functions

Declared in: be/interface/InterfaceDefs.h
Library: libbe.so
Summary: *more...*

This section describes the global mouse and keyboard functions.

Mouse Functions

- get_click_speed() see set_click_speed()
- get_mouse_map() see set_mouse_map()
- get_mouse_speed() see set_mouse_map()
- get_mouse_acceleration() see set_mouse_acceleration()
- get_mouse_type() see set_mouse_map()

- set_click_speed() , get_click_speed()

Declared in: be/interface/InterfaceDefs.h

status_t **set_click_speed**(bigtime_t *interval*)
status_t **get_click_speed**(bigtime_t **interval*)

These functions set and report the timing for multiple-clicks. For successive mouse-down events to count as a multiple-click, they must occur within the *interval* set by **set_click_speed()** and provided by **get_click_speed()**. The interval is measured in microseconds; it's usually set by the user in the Mouse preferences application. The smallest possible interval is 100,000 microseconds (0.1 second).

If successful, these functions return **B_OK**; if unsuccessful, they return an error code, which may be just **B_ERROR**.

- set_mouse_map() , get_mouse_map() , set_mouse_type() , get_mouse_type() , set_mouse_speed() , get_mouse_speed()

Declared in: be/interface/InterfaceDefs.h

status_t **set_mouse_map**(mouse_map **map*)
status_t **get_mouse_map**(mouse_map **map*)
status_t **set_mouse_type**(int32 *numButtons*)
status_t **get_mouse_type**(int32 **numButtons*)
status_t **set_mouse_speed**(int32 *speed*)
status_t **get_mouse_speed**(int32 **speed*)
status_t **set_mouse_acceleration**(int32 *acceleration*)
status_t **get_mouse_acceleration**(int32 **acceleration*)

These functions configure the mouse and supply information about the current configuration. The configuration should usually be left to the user and the Mouse preferences application.

`set_mouse_map()` maps the buttons of the mouse to their roles in the user interface, and `get_mouse_map()` writes the current map into the variable referred to by *map*. The `mouse_map` structure has a field for each button on a three-button mouse:

uint32 **`left`**
The button on the left of the mouse

uint32 **`right`**
The button on the right of the mouse

uint32 **`middle`**
The button in the middle, between the other two buttons

Each field is set to one of the following constants:

`B_PRIMARY_MOUSE_BUTTON`
`B_SECONDARY_MOUSE_BUTTON`
`B_TERTIARY_MOUSE_BUTTON`

The same role can be assigned to more than one physical button. If all three buttons are set to `B_PRIMARY_MOUSE_BUTTON`, they all function as the primary button; if two of them are set to `B_SECONDARY_MOUSE_BUTTON`, they both function as the secondary button; and so on.

`set_mouse_type()` informs the system of how many buttons the mouse actually has. If it has two buttons, only the `left` and `right` fields of the `mouse_map` are operative. If it has just one button, only the `left` field is operative. `set_mouse_type()` writes the current number of buttons into the variable referred to by *numButtons*.

`set_mouse_speed()` sets the speed of the mouse—the rate at which the cursor image moves on-screen relative to the actual speed at which the user moves the mouse on its pad. A *speed* value of 0 is the slowest movement rate. The maximum rate is 20, though even 10 is too fast for most users. `get_mouse_speed()` writes the current speed into the variable referred to by *speed*.

`set_mouse_acceleration()` sets the mouse's acceleration—the rate at which the cursor image gains and loses speed as the user begins and ceases moving the mouse. An *acceleration* value of 0 is the slowest movement rate. The maximum rate is 20, though even 10 is too fast for most users. `get_mouse_acceleration()` writes the current acceleration into the variable referred to by *acceleration*.

All six functions return `B_OK` if successful, and an error code, typically `B_ERROR`, if not.

Keyboard Functions

get_key_info()

Declared in: be/interface/InterfaceDefs.h

```
status_t get_key_info(key_info *keyInfo)
```

Writes information about the state of the keyboard into the `key_info` structure referred to by *keyInfo*. This function lets you get information about the keyboard in the absence of `B_KEY_DOWN` messages. The `key_info` structure has just two fields:

uint32 **modifiers**
A mask indicating which modifier keys are down and which keyboard locks are on.

uint8 **key_states** [16]
A bit array that records the state of all the keys on the keyboard, and all the keyboard locks. This array works identically to the "states" array passed in a key-down message. See "Key States" in the **Keyboard Information** appendix for information on how to read information from the array.

`get_key_info()` returns `B_OK` if it was able to get the requested information, and `B_ERROR` if the return results are unreliable.

See also: `BView::KeyDown()`, the **Keyboard Information** appendix, `modifiers()`

---

get_key_map()

Declared in: be/interface/InterfaceDefs.h

```
void get_key_map(key_map **keys, char **chars)
```

Provides a pointer to a copy of the system key map—the structure that describes the role of each key on the keyboard. The pointers returned by the function are yours; you must `free()` them when you're finished with them.

In versions of the BeOS before Release 4, the pointers used to belong to the operating system. Now they're yours to do with as you please. Please update your applications as necessary to avoid leaking memory.

Through the Keymap preferences application, users can configure the keyboard to their liking. The user's preferences are stored in a file (**Key_map** within the `B_USER_SETTINGS_DIRECTORY`, returned by the `find_directory()` function). When the machine reboots, the key map is read from this file. If the file doesn't exist, the original map encoded in the Application Server is used.

The `key_map` structure contains a large number of fields, but it can be broken down into these six parts:

- A version number.
- A series of fields that determine which keys will function as modifier keys—such as Shift, Control, or Num Lock.
- A field that sets the initial state of the keyboard locks in the default key map.
- A series of ordered tables that assign character values to keys. Except for a handful of modifier keys, all keys are mapped to characters, though they may not be mapped for all modifier combinations.
- A series of tables that locate the dead keys for diacritical marks and determine how a combination of a dead key plus another key is mapped to a particular character.
- A set of masks that determine which modifier keys are required for a key to be considered dead.

The following sections describe the parts of the `key_map` structure.

Version

The first field of the key map is a version number:

uint32 **`version`**
An internal identifier for the key map.

The version number doesn't change when the user configures the keyboard, and shouldn't be changed programmatically either. You can ignore it.

Modifiers

Modifier keys set states that affect other user actions on the keyboard and mouse. Eight modifier states are defined—Shift, Control, Option, Command, Menu, Caps Lock, Num Lock, and Scroll Lock. These states are discussed under "Modifier Keys" in the **Keyboard Information** appendix. They fairly closely match the key caps found on a Macintosh keyboard, but only partially match those on a standard PC keyboard—which generally has a set of Alt(ernate) keys, rarely Option keys, and only sometimes Command and Menu keys. Because of these differences, the mapping of keys to modifiers is the area of the key map most open to the user's personal judgement and taste, and consequently to changes in the default configuration.

Since two keys, one on the left and one on the right, can be mapped to the Shift, Control, Option, and Command modifiers, the keyboard can have as many as twelve modifier keys. The `key_map` structure has one field for each key:

uint32 **`caps_key`**
The key that functions as the Caps Lock key; by default, this is the key labeled "Caps Lock," key 0x3b.

uint32 **`scroll_key`**
The key that functions as the Scroll Lock key; by default, this is the key labeled "Scroll Lock," key 0x0f.

uint32 **`num_key`**
The key that functions as the Num Lock key; by default, this is the key labeled "Num Lock," key 0x22.

uint32 **`left_shift_key`**
A key that functions as a Shift key; by default, this is the key on the left labeled "Shift," key 0x4b.

uint32 **`right_shift_key`**
Another key that functions as a Shift key; by default, this is the key on the right labeled "Shift," key 0x56.

uint32 **`left_command_key`**
A key that functions as a Command key; by default, this is key 0x5d, sometimes labeled "Alt."

uint32 **`right_command_key`**
Another key that functions as a Command key; by default, this is key 0x5f, sometimes labeled "Alt."

uint32 **`left_control_key`**
A key that functions as a Control key; by default, this is the key labeled "Control" on the left, key 0x5c.

uint32 **`right_control_key`**
Another key that functions as a Control key; by default on keyboards that have Option keys, this key is the key labeled "Control" on the right, key 0x60. For keyboards that don't have Option keys, this field is unmapped (its value is 0); key 0x60 is used as an Option key.

uint32 **`left_option_key`**
A key that functions as an Option key; by default, this is key 0x66, which has different labels on different keyboards—"Option," "Command," or a Windows symbol. This key doesn't exist on, and therefore isn't mapped for, a standard 101-key keyboard.

uint32 **`right_option_key`**
A key that functions as an Option key; by default, this is key 0x67, which has different labels on different keyboards—"Option," "Command," or a Windows symbol. For keyboards without this key, the field is mapped to the key labeled "Control" on the right, key 0x60.

uint32 **`menu_key`**
A key that initiates keyboard navigation of the menu hierarchy; by default, this is the key labeled with a menu symbol, key 0x68. This key doesn't exist on, and therefore isn't mapped for, a standard 101-key keyboard.

Each field names the key that functions as that modifier. For example, when the user holds down the key whose code is set in the **`right_option_key`** field, the **`B_OPTION_KEY`** and

B_RIGHT_OPTION_KEY bits are turned on in the modifiers mask that the modifiers() function returns. When the user then strikes a character key, the B_OPTION_KEY state influences the character that's generated.

If a modifier field is set to a value that doesn't correspond to an actual key on the keyboard (including 0), that field is not mapped. No key fills that particular modifier role.

Keyboard locks

One field of the key map sets initial modifier states:

uint32 **lock_settings**
A mask that determines which keyboard locks are turned on when the machine reboots or when the default key map is restored.

The mask can be 0 or may contain any combination of these three constants:

B_CAPS_LOCK
B_SCROLL_LOCK
B_NUM_LOCK

It's 0 by default; there are no initial locks.

Altering the lock_settings field has no effect unless the altered key map is made the default.

Character maps

The principal job of the key map is to assign character values to keys. This is done in a series of nine tables:

int32 **control_map** [128]
The characters that are produced when a Control key is down but both Command keys are up.

int32 **option_caps_shift_map** [128]
The characters that are produced when Caps Lock is on and both a Shift key and an Option key are down.

int32 **option_caps_map** [128]
The characters that are produced when Caps Lock is on and an Option key is down.

int32 **option_shift_map** [128]
The characters that are produced when both a Shift key and an Option key are down.

int32 **option_map** [128]
The characters that are produced when an Option key is down.

int32 **caps_shift_map** [128]
The characters that are produced when Caps Lock is on and a Shift key is down.

int32 **caps_map** [128]
The characters that are produced when Caps Lock is on.

int32 **shift_map** [128]
The characters that are produced when a Shift key is down.

int32 **normal_map** [128]
The characters that are produced when none of the other tables apply.

Each of these tables is an array of 128 offsets into another array, the *chars* array of Unicode UTF-8 character encodings. **get_key_map()** provides a pointer to the *chars* array as its second argument.

Key codes are used as indices into the character tables. The offset stored at any particular index maps a character to that key. For example, the code assigned to the *M* key is 0x52; at index 0x52 in the **option_caps_map** is an offset; at that offset in the *chars* array, you'll find the character that's mapped to the *M* key when an Option key is held down and Caps Lock is on.

This indirection—an index to an offset to a character—is required because characters are encoded as Unicode UTF-8 strings. Character values of 127 or less (7-bit ASCII) are just a single byte, but UTF-8 takes two, three, or (rarely) four bytes to encode values over 127.

The *chars* array represents each character as a Pascal string—the first byte in the string tells how many other bytes the string contains. For example, the string for the trademark symbol ((TM)) looks like this:

```
x03xE2x84xA2
```

The first byte (x03) indicates that Unicode UTF-8 takes 3 bytes to represent the trademark symbol, and those bytes follow (xE2x84xA2). Pascal strings are not null-terminated.

Here's an example showing you how to decode the character tables. This sample prints out a simple chart of the **normal_map**, **shift_map**, **option_map**, and **option_shift_map** characters:

```
#include <interface/InterfaceDefs.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void print_key( char *chars, int32 offset )
{
   int size = chars[offset++];

   switch( size ) {
   case 0:
      // Not mapped
      printf( "N/A" );
      break;
```

```
   case 1:
      // 1-byte UTF-8/ASCII character
      printf( "%c", chars[offset] );
      break;

   default:
      // 2-, 3-, or 4-byte UTF-8 character
      {
         char str[5];
         int i = (size <= 4) ? size : 4;
         strncpy( str, &(chars[offset]), i );
         str[i] = '0';
         printf( "%s", str );
      }
      break;
   }

   printf( "t" );
}

int main( void )
{
   // Get the current key map.
   key_map *keys;
   char *chars;
   get_key_map( &keys, &chars );

   // Print a chart of the normal, shift, option, and option+shift
   // keys.
   printf( "Key #tNormaltShifttOptiontOption+Shiftn" );
   for( int idx = 0; idx < 128; idx++ ) {
      printf( " %3dt", idx );
      print_key( chars, keys->normal_map[idx] );
      print_key( chars, keys->shift_map[idx] );
      print_key( chars, keys->option_map[idx] );
      print_key( chars, keys->option_shift_map[idx] );
      printf( "n" );
   }

   // Free our copy of the key map.
   free( chars );
   free( keys );

   return EXIT_SUCCESS;
}
```

The character map tables are ordered. Values from the first applicable table are used, even if another table might also seem to apply. For example, if Caps Lock is on and a Control key is down (and both Command keys are up), the **control_map** array is used, not **caps_map**. If a Shift key is down and Caps Lock is on, the **caps_shift_map** is used, not **shift_map** or **caps_map**.

Notice that the last eight tables (all except **control_map**) are paired, with a table that names the Shift key (..._shift_map) preceding an equivalent table without Shift:

- **option_caps_shift_map** is paired with **option_caps_map**,
- **option_shift_map** with **option_map**,
- **caps_shift_map** with **caps_map**, and
- **shift_map** with **normal_map**.

These pairings are important for a special rule that applies to keys on the numerical keypad when Num Lock is on:

- If the Shift key is down, the non-Shift table is used.
- However, if the Shift key is *not* down, the Shift table is used.

In other words, Num Lock inverts the Shift and non-Shift tables for keys on the numerical keypad.

Not every key needs to be mapped to a character. If the *chars* array has a 0-length string for a key, the key is not mapped to a character (given the particular modifier states the table represents). Generally, modifier keys are not mapped to characters, but all other keys are, at least for some tables. Key-down events are not generated for unmapped keys.

Dead keys

Next are the tables that map combinations of keys to single characters. The first key in the combination is "dead"—it doesn't produce a key-down event until the user strikes another character key. When the user hits the second key, one of two things will happen: If the second key is one that can be used in combination with the dead key, a single key-down event reports the combination character. If the second key doesn't combine with the dead key, two key-down events occur, one reporting the dead-key character and one reporting the second character.

There are five dead-key tables:

int32 **acute_dead_key** [32]
The table for combining an acute accent (xab) with other characters.

int32 **grave_dead_key** [32]
The table for combining a grave accent (Q) with other characters.

int32 **circumflex_dead_key** [32]
The table for combining a circumflex (xf6) with other characters.

int32 **dieresis_dead_key** [32]
The table for combining a dieresis (xac) with other characters.

int32 **tilde_dead_key** [32]
The table for combining a tilde (xf7) with other characters

The tables are named after diacritical marks that can be placed on more than one character. However, the name is just a mnemonic; it means nothing. The contents of the table determine what the dead key is and how it combines with other characters. It would be possible, for example, to remap the **tilde_dead_key** table so that it had nothing to do with a tilde.

Each table consists of a series of up to 16 offset pairs—where, as in the case of the character maps, each offset picks a character from the *chars* character array. The first character in the pair is the one that must be typed immediately after the dead key. The second character is the resulting character, the character that's produced by the combination of the dead key plus the first character in the pair. For example, if the first character is 'o', the second might be 'ô'—meaning that the combination of a dead key plus the character 'o' produces a circumflexed 'ô'.

The character pairs for the default **grave_dead_key** array look something like this:

```
' ', ''',
'A', 'À',
'E', 'È',
'I', 'Ì',
'O', 'Ò',
'U', 'Ù',
'a', 'à',
'e', 'è',
'i', 'ì',
'o', 'ò',
'u', 'ù',
. . .
```

By convention, the first offset in each array is to the **B_SPACE** character and the second is to the dead-key character itself. This pair does double duty: It states that the dead key plus a space yields the dead-key character, and it also names the dead key. The system understands what the dead key is from the second offset in the array.

Character tables for dead keys

As mentioned above, for a key to be dead, it must be mapped to the character picked by the second offset in a dead-key array. However, it's not typical for every key that's mapped to the character to be dead. Usually, there's a requirement that the user must hold down certain modifier keys (often the Option key). In other words, a key is dead only if selected character-map tables map it to the requisite character.

Five additional fields of the **key_map** structure specify what those character-map tables are—which modifiers are required for each of the dead keys:

uint32 **acute_tables**
The character tables that cause a key to be dead when they map it to the second character in the **acute_dead_key** array.

uint32 **grave_tables**
The character tables that cause a key to be dead when they map it to the second character in the **grave_dead_key** array.

uint32 **circumflex_tables**
The character tables that cause a key to be dead when they map it to the second character in the **circumflex_dead_key** array.

uint32 **dieresis_tables**
The character tables that cause a key to be dead when they map it to the second character in the **dieresis_dead_key** array.

uint32 **tilde_tables**
The character tables that cause a key to be dead when they map it to the second character in the **tilde_dead_key** array.

Each of these fields contains a mask formed from the following constants:

B_CONTROL_TABLE	**B_CAPS_SHIFT_TABLE**
B_OPTION_CAPS_SHIFT_TABLE	**B_CAPS_TABLE**
B_OPTION_CAPS_TABLE	**B_SHIFT_TABLE**
B_OPTION_SHIFT_TABLE	**B_NORMAL_TABLE**
B_OPTION_TABLE	

The mask designates the character-map tables that permit a key to be dead. For example, if the mask for the **grave_tables** field is,

```
B_OPTION_TABLE | B_OPTION_CAPS_SHIFT_TABLE
```

a key would be dead whenever either of those tables mapped the key to the character of the second offset in the **grave_dead_key** array ('Q' in the example above). A key mapped to the same character by another table would not be dead.

See also: **get_key_info()**, **modifiers()**, the **Keyboard Information** appendix, **set_modifier_key()**

---

get_key_repeat_delay() see set_key_repeat_rate()
get_key_repeat_rate() see set_key_repeat_rate()

---

get_keyboard_id()

Declared in: be/interface/InterfaceDefs.h

status_t **get_keyboard_id**(uint16 *_id_)

Obtains the keyboard identifier from the Application Server and device driver and writes it into the variable referred to by *id*. This number reveals what kind of keyboard is currently attached to the computer.

The identifier for the standard 101-key PC keyboard—and for keyboards with a similar set of keys—is 0x83ab.

If unsuccessful for any reason, `get_keyboard_id()` returns `B_ERROR`. If successful, it returns `B_OK`.

---

modifiers()

Declared in: be/interface/InterfaceDefs.h

uint32 **modifiers**(void)

Returns a mask that has a bit set for each modifier key the user is holding down and for each keyboard lock that's set. The mask can be tested against these constants:

B_SHIFT_KEY	**B_COMMAND_KEY**	**B_CAPS_LOCK**
B_CONTROL_KEY	**B_MENU_KEY**	**B_SCROLL_LOCK**
B_OPTION_KEY		**B_NUM_LOCK**

No bits are set (the mask is 0) if no locks are on and none of the modifiers keys are down.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be further tested against these constants:

B_LEFT_SHIFT_KEY	**B_RIGHT_SHIFT_KEY**
B_LEFT_CONTROL_KEY	**B_RIGHT_CONTROL_KEY**
B_LEFT_OPTION_KEY	**B_RIGHT_OPTION_KEY**
B_LEFT_COMMAND_KEY	**B_RIGHT_COMMAND_KEY**

By default, the keys closest to the space bar function as Command keys, no matter what their labels on particular keyboards. If a keyboard doesn't have Option keys (for example, a standard 101-key keyboard), the key on the right labeled "Control" functions as the right Option key, and only the left "Control" key is available to function as a Control modifier. However, users can change this configuration with the **/bin/keymap** application.

---

set_key_repeat_rate() , get_key_repeat_rate() , set_key_repeat_delay() , get_key_repeat_delay()

Declared in: be/interface/InterfaceDefs.h

status_t **set_key_repeat_rate**(int32 *rate*)

status_t **get_key_repeat_rate**(int32 *_rate_)
status_t **set_key_repeat_delay**(bigtime_t _delay_)
status_t **get_key_repeat_delay**(bigtime_t *_delay_)

These functions set and report the timing of repeating keys. When the user presses a character key on the keyboard, it produces an immediate `B_KEY_DOWN` message. If the user continues to hold the key down, it will, after an initial delay, continue to produce messages at regularly spaced intervals—until the user releases the key or presses another key. The delay and the spacing between messages are both preferences the user can set with the Keyboard application.

`set_key_repeat_rate()` sets the number of messages repeating keys produce per second. For a standard PC keyboard, the *rate* can be as low as 2 and as high as 30; `get_key_repeat_rate()` writes the current setting into the integer that *rate* refers to.

`set_key_repeat_delay()` sets the length of the initial delay before the key begins repeating. Acceptable values are 250,000, 500,000, 750,000 and 1,000,000 microseconds (.25, .5, .75, and 1.0 second); `get_key_repeat_delay()` writes the current setting into the variable that *delay* points to.

All four functions return `B_OK` if they successfully communicate with the Application Server, and `B_ERROR` if not. It's possible for the `set...()` functions to communicate with the server but not succeed in setting the *rate* or *delay* (for example, if the *delay* isn't one of the listed four values).

---

set_keyboard_locks()

Declared in: be/interface/InterfaceDefs.h

void **set_keyboard_locks**(uint32 _modifiers_)

Turns the keyboard locks—Caps Lock, Num Lock, and Scroll Lock—on and off. The keyboard locks that are listed in the *modifiers* mask passed as an argument are turned on; those not listed are turned off. The mask can be 0 (to turn off all locks) or it can contain any combination of the following constants:

B_CAPS_LOCK
B_NUM_LOCK
B_SCROLL_LOCK

See also: `get_key_map()`, `modifiers()`

---

set_modifier_key()

Declared in: be/interface/InterfaceDefs.h

void **set_modifier_key(**uint32 *modifier*, uint32 *key***)**

Maps a *modifier* role to a particular *key* on the keyboard, where *key* is a key identifier and *modifier* is one of the these constants:

B_CAPS_LOCK	**B_LEFT_SHIFT_KEY**	**B_RIGHT_SHIFT_KEY**
B_NUM_LOCK	**B_LEFT_CONTROL_KEY**	**B_RIGHT_CONTROL_KEY**
B_SCROLL_LOCK	**B_LEFT_OPTION_KEY**	**B_RIGHT_OPTION_KEY**
B_MENU_KEY	**B_LEFT_COMMAND_KEY**	**B_RIGHT_COMMAND_KEY**

The *key* in question serves as the named modifier key, unmapping any key that previously played that role. The change remains in effect until the default key map is restored. In general, the user's preferences for modifier keys—expressed in the Keymap application—should be respected.

Modifier keys can also be mapped by calling **get_key_map()** and altering the **key_map** structure directly. This function is merely a convenient alternative for accomplishing the same thing. (It's currently not possible to alter the key map; **get_key_map()** looks at a copy.)

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Input Server Table of Contents

- The Input Server
- BInputDevice
- BInputServerDevice
- BInputServerFilter
- BInputServerMethod
- Input Server Messages
- Input Functions
- Input Server Structures and Constants

The Be Book **The Input Server Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Input Server: Master Index

+

+116444	BInputDevice
+242049	BInputDevice
+607719	BInputServerFilter
+724878	BInputDevice
+787205	BInputDevice
+811648	BInputDevice
+845847	BInputDevice
+908287	BInputDevice

-

-116444	anon
-242049	anon
-724878	anon
-787205	anon
-811648	anon
-845847	anon
-908287	anon

=

=116444	anon
=242049	anon
=724878	anon
=787205	anon
=811648	anon
=845847	anon
=908287	anon

A

Absolute Locations	BInputServerDevice
acute_dead_key	Input Functions
acute_tables	Input Functions
App and Input Events	The Input Server

B

Building	The Input Server

C

C Functions	BInputDevice
caps_key	Input Functions
B_CAPS_LOCK	Input Functions
B_CAPS_LOCK	Input Functions
caps_map	Input Functions
caps_shift_map	Input Functions

B_CAPS_SHIFT_TABLE	Input Functions
B_CAPS_TABLE	Input Functions
Character maps	Input Functions
Character tables for dead keys	Input Functions
circumflex_dead_key	Input Functions
circumflex_tables	Input Functions
B_CLICK_SPEED_CHANGED	Input Server Messages
B_COMMAND_KEY	Input Functions
Constants	Input Server Structures and Constants
Constructor and Destructor	BInputDevice
Constructor and Destructor	BInputServerDevice
Constructor and Destructor	BInputServerFilter
Constructor and Destructor	BInputServerMethod
Control()	BInputDevice
Control()	BInputServerDevice
Control	Input Server Structures and Constants
B_CONTROL_KEY	Input Functions
control_map	Input Functions
B_CONTROL_TABLE	Input Functions
Creating and Registering	BInputServerDevice
Creating	BInputServerFilter
Creating	BInputServerMethod

D

Dead keys	Input Functions
Device	Input Server Structures and Constants
Device Monitoring	Input Server Messages
Device State	BInputServerDevice
Device Types and Control Messages	BInputServerDevice
Device Types	Input Server Structures and Constants
dieresis_dead_key	Input Functions
dieresis_tables	Input Functions
Drivers and Input Devices	The Input Server
Dynamic Devices	BInputServerDevice

E

EnqueueMessage()	BInputServerDevice
EnqueueMessage()	BInputServerMethod
Eraser Mode	The Input Server

F

Filter()	BInputServerFilter
find_input_device()	BInputDevice

G

GetScreenRegion()	BInputServerFilter
get_click_speed()	Input Functions
get_input_devices()	BInputDevice
get_key_info()	Input Functions
get_key_map()	Input Functions
get_key_repeat_delay()	Input Functions
get_key_repeat_rate()	Input Functions
get_keyboard_id()	Input Functions
get_mouse_map()	Input Functions
get_mouse_speed()	Input Functions
get_mouse_type()	Input Functions
grave_dead_key	Input Functions
grave_tables	Input Functions

H

Hook Functions	BInputServerDevice
Hook Functions	BInputServerFilter
Hook Functions	BInputServerMethod

I

InitCheck()	BInputServerDevice
InputCheck()	BInputServerFilter
BInputDevice	BInputDevice

BInputDevice()	BInputDevice
~BInputDevice()	BInputDevice
Input Device Control Messages	BInputServerDevice
Input Device Control Messages	Input Server Messages
Input Device Event Messages	Input Server Messages
Input Functions	Input Functions
Input Functions	Input Functions
Input	Input Server Structures and Constants
Input Method Events	BInputServerMethod
Input Method Events	Input Server Messages
Input Server and You	The Input Server
BInputServerDevice	BInputServerDevice
BInputServerDevice()	BInputServerDevice
~BInputServerDevice()	BInputServerDevice
BInputServerFilter	BInputServerFilter
BInputServerFilter()	BInputServerFilter
~BInputServerFilter()	BInputServerFilter
Input Server Messages	Input Server Messages
Input Server Messages	Input Server Messages
BInputServerMethod	BInputServerMethod

BInputServerMethod()	BInputServerMethod
~BInputServerMethod()	BInputServerMethod
Input Server Structures and Constants	Input Server Structures and Constants
Input Server Structures and Constants	Input Server Structures and Constants
The Input Server	The Input Server
The Input Server	The Input Server
B_INPUT_DEVICE_ADDED	Input Server Messages
input_device_ref	Input Server Structures and Constants
B_INPUT_DEVICE_REMOVED	Input Server Messages
B_INPUT_DEVICE_STARTED	Input Server Messages
B_INPUT_DEVICE_STOPPED	Input Server Messages
B_INPUT_METHOD_CHANGED	Input Server Messages
B_INPUT_METHOD_LOCATION_REQUEST	Input Server Messages
B_INPUT_METHOD_STARTED	Input Server Messages
B_INPUT_METHOD_STOPPED	Input Server

	Messages
Installing an Input Device	BInputServerDevice
Installing an Input Filter	BInputServerFilter
Installing an Input Method	BInputServerMethod
Installing	The Input Server
IsRunning()	BInputDevice

K

B_KEY_LOCKS_CHANGED	Input Server Messages
B_KEY_MAP_CHANGED	Input Server Messages
B_KEY_REPEAT_DELAY_CHANGED	Input Server Messages
B_KEY_REPEAT_RATE_CHANGED	Input Server Messages
key_states	Input Functions
Keyboard Device Control Messages	Input Server Messages
Keyboard Device Event Messages	Input Server Messages
Keyboard Functions	Input Functions
Keyboard Functions	Input Functions
Keyboard locks	Input Functions

L

left	Input Functions
B_LEFT_COMMAND_KEY	Input Functions
B_LEFT_CONTROL_KEY	Input Functions
B_LEFT_OPTION_KEY	Input Functions

B_LEFT_SHIFT_KEY	Input Functions
Loading	The Input Server
lock_settings	Input Functions

M

Member Functions	BInputDevice
Member Functions	BInputServerDevice
Member Functions	BInputServerFilter
Member Functions	BInputServerMethod
B_MENU_KEY	Input Functions
B_MENU_KEY	Input Functions
Messages from Input Methods	The Input Server
Messages	Input Server Structures and Constants
MethodActivated()	BInputServerMethod
Method	Input Server Structures and Constants
Mice and Tablets	The Input Server
middle	Input Functions
Modifiers	Input Functions
modifiers()	Input Functions
Mouse Functions	Input Functions
B_MOUSE_MAP_CHANGED	Input Server Messages
B_MOUSE_SPEED_CHANGED	Input Server Messages
B_MOUSE_TYPE_CHANGED	Input Server Messages

N

Name()	BInputDevice
Node Monitor Messages	BInputServerDevice
normal_map	Input Functions
B_NORMAL_TABLE	Input Functions
Notifications	Input Server Structures and Constants
Now where?	BInputServerDevice
num_key	Input Functions
B_NUM_LOCK	Input Functions
B_NUM_LOCK	Input Functions

O

Operations	Input Server Structures and Constants
option_caps_map	Input Functions
option_caps_shift_map	Input Functions
B_OPTION_CAPS_SHIFT_TABLE	Input Functions
B_OPTION_CAPS_TABLE	Input Functions
B_OPTION_KEY	Input Functions
option_map	Input Functions
option_shift_map	Input Functions
B_OPTION_SHIFT_TABLE	Input Functions
B_OPTION_TABLE	Input Functions

Other Useful Information	BInputServerDevice

P

Pointing Device Control Messages	Input Server Messages
Pointing Device Event Messages	Input Server Messages
Pointing Devices	BInputServerDevice
Precision Position Information	The Input Server
Pressure	The Input Server
B_PRIMARY_MOUSE_BUTTON	Input Functions

R

RegisterDevices()	BInputServerDevice
Relative Locations	BInputServerDevice
right	Input Functions
B_RIGHT_COMMAND_KEY	Input Functions
B_RIGHT_COMMAND_KEY	Input Functions
B_RIGHT_CONTROL_KEY	Input Functions
B_RIGHT_CONTROL_KEY	Input Functions
B_RIGHT_OPTION_KEY	Input Functions
B_RIGHT_OPTION_KEY	Input Functions
B_RIGHT_SHIFT_KEY	Input Functions
B_RIGHT_SHIFT_KEY	Input Functions

S

scroll_key	Input Functions
B_SCROLL_LOCK	Input Functions
B_SCROLL_LOCK	Input Functions
B_SECONDARY_MOUSE_BUTTON	Input Functions
SetIcon()	BInputServerMethod
SetMenu()	BInputServerMethod
SetName()	BInputServerMethod
set_click_speed()	Input Functions
set_key_repeat_delay()	Input Functions
set_key_repeat_rate()	Input Functions
set_keyboard_locks()	Input Functions
set_modifier_key()	Input Functions
set_mouse_map()	Input Functions
set_mouse_speed()	Input Functions
set_mouse_type()	Input Functions
B_SHIFT_KEY	Input Functions
shift_map	Input Functions
B_SHIFT_TABLE	Input Functions
Start()	BInputDevice
Start()	BInputServerDevice
StartMonitoringDevice()	BInputServerDevice
Starting and Sending Messages	BInputServerDevice
Stop()	BInputDevice
Stop()	BInputServerDevice
StopMonitoringDevice()	BInputServerDevice
Structures	Input Server Structures and

	Constants
Supporting Input Methods in Views	The Input Server
SystemShuttingDown()	BInputServerDevice

T

B_TERTIARY_MOUSE_BUTTON	Input Functions
tilde_dead_key	Input Functions
tilde_tables	Input Functions
Tilt	The Input Server
Type()	BInputDevice

U

UnregisterDevices()	BInputServerDevice

V

Version	Input Functions
version	Input Functions

W

watch_input_devices()	BInputDevice

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputDevice

Derived from: *none*
Declared in: be/interface/Input.h
Library: libbe.so
Allocation: By the system only. See **`find_input_device()`**.
more...

A BInputDevice object is a "downstream" representation of an Input Server device, such as a mouse or a keyboard, within a "regular" application. The BInputDevice can **`Start()`** and **`Stop()`** the device it represents, and can send it *input device control messages* through its **`Control()`** function.

You never create BInputDevice objects yourself; instead, you ask the system to return one or more instances to you through the **`find_input_device()`** or **`get_input_devices()`** functions. Alternatively, you can work without an object by invoking the static versions of **`Start()`**, **`Stop()`**, and **`Control()`**. Note, however, that the static functions control *all* devices of a given type, whereas a BInputDevice instance can talk to a specific device.

BInputDevice objects don't live in the Input Server—they're used in "normal" applications as a means to control an Input Server device add-on.

The BInputDevice object is provided, primarily, to let an application talk to a custom input device.

You never subclass BInputDevice.

Constructor and Destructor

BInputDevice()

The constructor is private. Use `find_input_device()` or `get_input_devices()` to retrieve a BInputDevice instance.

~BInputDevice()

```
~BInputDevice()
```

Deletes the BInputDevice object. Deleting this object doesn't affect the device that it represents.

Member Functions

Control()

```
status_t Control(uint32 code, BMessage *message)
static status_t Control(input_device_type type,
    uint32 code,
    BMessage *message)
```

Sends an input device control message to the object's input device or, in the static version, to all devices of the given type, where, *type* can be **B_POINTING_DEVICE**, **B_KEYBOARD_DEVICE**, or **B_UNDEFINED_DEVICE**. Input devices receive these messages in their **Control()** function.

The control message is described by the *code* value; it can be supplemented or refined by *message*. For example, *code* can indicate that a certain parameter should be set, and *message* can supply the requested value.

In general, you only use this function to send custom messages to a (custom) device. You never use it to send messages to a Be-defined input device since the messages that these devices respond to are covered by Be-defined functions. See "Input Device Control Messages" for a list of the messages that the Be-defined devices respond to, the functions that cover them, and the German women who love them.

Start()

Name() , Type()

```
const char *Name(void) const
input_device_type Type(void) const
```

Name()**Start()** returns a pointer to the input device's name. The name, which is set when the device is registered, is meant to be human-readable and appropriate for use as the label of a UI element (such as a menu field). Device names are not unique.

Type() returns the input device's type, one of **B_POINTING_DEVICE**, **B_KEYBOARD_DEVICE**, and **B_UNDEFINED_DEVICE**.

Start() , Stop() , IsRunning()

```
status_t Start(void)
status_t Start(void)
bool IsRunning(void) const
```

static status_t **Start(**input_device_type *type***)**
static status_t **Stop(**input_device_type *type***)**

Start() tells the object's input device to start generating events; **Stop()** tells it to stop generating events. **IsRunning()** returns true if the device is currently generating events (i.e. if it has been started and hasn't been stopped).

The static versions of **Start()** and **Stop()** start and stop all devices of the given *type*.

RETURN CODES

- B_OK. The device is now started (**Start()**) or stopped (**Stop()**)—even if the device was already started or stopped.
- **B_ERROR**. The device couldn't be found.

The Input Server tells a device to start and stop without asking the device if the operation was successful. This means, for example, that **Start()** can return **B_OK** (and **IsRunning()** can return **true**) even if the device isn't really running. For the Be-provided devices this isn't an issue—starting and stopping always succeeds (as long as the device exists).

- Start()
- Type() see Name()

C Functions

find_input_device() , get_input_devices()

BInputDevice* **find_input_device(**const char **name***)*
status_t **get_input_devices(**BList ***list***)

These functions get BInputDevice objects for you.

find_input_device() creates and hands you a BInputDevice object that represents the Input Server device registered as *name*. If *name* is invalid, the function returns **NULL**. The caller is responsible for deleting the object. Note that **find_input_device()** returns a new BInputDevice object for each (valid) call, even if you ask for the same device more than once.

get_input_devices() creates a new BInputDevice object for each registered device, and puts the objects in your *list* argument. *list* must already be allocated, and is automatically emptied by the function (even if the function fails). If the function succeeds, the caller owns the contents, and needs to delete the items in the list:

```
#include <interface/Input.h>
#include <support/List.h>

...

static bool del_InputDevice( void *ptr )
{
   if( ptr ) {
      BInputDevice *dev = (BInputDevice *)ptr;
      delete dev;

      return false;
   }

   return true;
}

...

void SomeFunc( void )
{
   // Get a list of all input devices.
   BList list_o_devices;

   status_t retval = get_input_devices( &list_o_devices );
   if( retval != B_OK ) return;
```

```
    // Do something with the input devices.
    ...

    // Dispose of the device list.
    list_o_devices.DoForEach( del_InputDevice );
    list_o_devices.MakeEmpty();
}
```

RETURN CODES

`get_input_devices()` returns:

- `B_OK`. Success.
- `B_ERROR`. General failure.
- `B_BAD_PORT_ID`. Couldn't talk to the Input Server.
- `B_TIMED_OUT`. The Input Server is no longer on speaking terms with your application.
- `B_WOULD_BLOCK`. More trouble communicating with the Input Server.

---

watch_input_devices()

status_t **watch_input_devices(**BMessenger *target*, bool *start***)**

Tells the Input Server to start or stop watching (as *start* is **`true`** or **`false`**) for changes to the set of registered devices. Change notifications are sent to *target*. The set of messages that the Server may send are listed in **Input Server Messages**.

`watch_input_devices()` is not currently implemented.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputDevice Function Summary

Function Summary

Function	Description
IsRunning()	True if the input device is generating events
Start()	Tells the input device to start generating events
Stop()	Tells the input device to stop generating events

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

General Messages

B_ABOUT_REQUESTED

Purpose: Deliverable
Source: The system or your app.
Target: App-defined; typically **be_app**.
Hook: BApplication::AboutRequested() if the target is **be_app**.

The message should be assigned to an **About...** menu item, such that the message is sent when the user clicks the item. Your application is expected to put up an "About This Application" panel when it receives this message.

B_ACQUIRE_OVERLAY_LOCK , B_RELEASE_OVERLAY_LOCK

Source: A graphics driver.
Target: The team owning the overlay.

B_ACQUIRE_OVERLAY_LOCK is sent by a graphics driver when an overlay is acquired.
B_RELEASE_OVERLAY_LOCK is sent when the overlay is released.

B_APP_ACTIVATED

Purpose: Deliverable
Source: The system.
Target: **be_app**
Hook: BApplication::AppActivated()

Sent when an application becomes active or inactive.

Field	Type code	Description
"active"	B_BOOL_TYPE	**true** if the app has become active; otherwise **false**.

B_ARCHIVED_OBJECT

Purpose: Deliverable and format
Source: A dragged replicant, or your app.
Target: A (remote) application.
Hook: BShelf::CanAcceptReplicantMessage()

As a deliverable: The replicant system uses this message as a deliverable. If you're using BDragger and BShelf objects, the message is created and delivered for you. You can also simulate a dragged replicant by archiving a view, setting the archive message's command to B_ARCHIVED_OBJECT, and sending the message to a remote application. If the remote application

has a BShelf object, the BShelf will pick up the message (through a BMessageFilter) and pass it to the hook function.

To create a simulated replicant message, you call `Archive()` on the view that you want to replicate, and add (at least) the "add_on" field to the archive message.

See BShelf and BDragger for more information about replicants.

As a format: `B_ARCHIVED_OBJECT` should be used as the command constant for all archive messages. When you archive an object, the "class" field is automatically added to the archive message. All other fields must be added by your archiving code. See the BArchivable class for more information about archiving.

Field	Type code	Description
"class" [ ]	`B_STRING_TYPE`	An array of class names that gives the class hierarchy of the archived object.
"add_on"	`B_STRING_TYPE`	The signature of the library or application that knows how to create the archived object.
"be:add_on_version"	`B_INT32_TYPE`	The version of the add_on.
"be:load_each_time"	`B_BOOL_TYPE`	**`true`**: The add_on is loaded each time the object is unarchived. **`false`**: The add_on is loaded only if it isn't already loaded.
"be:unload_on_delete"	`B_BOOL_TYPE`	Is the add_on unloaded when the unarchived object is deleted?
"shelf_type" (replicants only; optional)	`B_STRING_TYPE`	The "type" of shelf that you want to have display the replicant. A shelf's type is its name, as assigned when it's created.

B_ARGV_RECEIVED

Source: The system.
Target: **`be_app`**
Hook: `BApplication::ArgvReceived()`

Forwards arguments that **(a)** the user passes while launching the app from the command line, or **(b)** are passed to `BRoster::Launch()`. Most apps treat command line arguments as filenames that should be opened. If the filename is relative (if it doesn't start with "/"), you should append it to the "cwd" field to reconstruct the full path.

Field	Type code	Description
"argc"	`B_INT32_TYPE`	The number of arguments.
"argv" [*argc*]	`B_STRING_TYPE`	The arguments.
"cwd"	`B_STRING_TYPE`	The path name of the current working directory.

B_CANCEL

Purpose: Deliverable
Source: The Application Kit.
Target: Application Server.

Used to cancel an ongoing operation. The Application Kit sends this message to the Application Server to cancel a shutdown if a window refuses to quit.

B_CLIPBOARD_CHANGED

Purpose: Deliverable
Source: The Application Server.
Target: Selected BMessenger.

If you've called BClipboard::`StartWatching()` to monitor a clipboard for changes, this message is sent to the specified BMessenger when the clipboard changes.

B_CONTROL_INVOKED

Purpose: Deliverable
Source: A BControl.
Target: Selected BMessenger.

This message is sent to the targeted messenger when a BControl-derived object is invoked.

B_INPUT_DEVICES_CHANGED

Purpose: Deliverable
Source: The Input Server.
Target: Add-on specified target.
Hook: `BInput::Control()`

This message is sent by the Input Server to send you notification when a device starts or stops, or when the set of registered devices changes. These messages are sent only if you've used the `watch_input_devices()` function to request such notification. The messages are sent to the target indicated in the function call.

Field	Type code	Description

"code"	`B_INT32_TYPE`	Operation code for the keyboard device control request being issued. One of:
		`B_INPUT_DEVICE_ADDED`
		`B_INPUT_DEVICE_STARTED`
		`B_INPUT_DEVICE_STOPPED`
		`B_INPUT_DEVICE_REMOVED`

"device"	`B_STRING_TYPE`	The name of the device. If this field doesn't exist, the device is identified by "type" instead.

"type"	B_INT32_TYPE	The device type. If this field doesn't exist, the "device" field identifies the device.
"message"	B_MESSAGE_TYPE	The control message. None of the BeOS standard messages use this field.

B_INPUT_METHOD_EVENT

Purpose: Deliverable
Source: The Input Server.
Target: Input method add-on's BLooper.

This message is sent by the Input Server to an input method add-on to let it know that a method event has occurred. The message's "be:opcode" field indicates which event has occurred.

- B_INPUT_METHOD_STARTED
- B_INPUT_METHOD_STOPPED
- B_INPUT_METHOD_CHANGED
- B_INPUT_METHOD_LOCATION_REQUEST

These are discussed in more detail in the Input Server chapter.

B_KEY_DOWN , B_KEY_UP , B_UNMAPPED_KEY_DOWN , B_UNMAPPED_KEY_UP

Source: The system.
Target: The focus view's BWindow.
Hook: `BView::KeyDown()` (`B_KEY_DOWN`)
`BView::KeyUp()` (`B_KEY_UP`)
(The `...UNMAPPED...` messages don't map to hook functions.)

B_KEY_DOWN is sent when the user presses (or holds down) a key that's mapped to a character; B_KEY_UP is sent when the user releases the key. B_UNMAPPED_KEY_DOWN and B_UNMAPPED_KEY_UP are sent if the key isn't mapped to a character. This doesn't include modifier keys, which are reported in the B_MODIFIERS_CHANGED message.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"key"	B_INT32_TYPE	The code for the physical key that was pressed. See <x> for the key map.
"be:key_repeat" (B_KEY_DOWN only)	B_INT32_TYPE	The "iteration number" of this key down. When the user holds the key down, successive messages are sent with increasing key repeat values. This field isn't present in the initial event;

		the first repeat message (i.e., the second key down message) has a key repeat value of 1.
"modifiers"	B_INT32_TYPE	The modifier keys that were in effect at the time of the event. See <x> for a list of values.
"states"	B_UINT8_TYPE	The state of all keys at the time of the event. See <x>.
"byte" [3] (B_KEY_DOWN and B_KEY_UP only)	B_INT8_TYPE	The UTF8 data that's generated
"bytes" (B_KEY_DOWN and B_KEY_UP only)	B_STRING_TYPE	The string that's generated. (The string usually contains a single character.)
"raw_char" (B_KEY_DOWN and B_KEY_UP only)	B_INT32_TYPE	Modifier-independent ASCII code for the character.

B_KEY_UP see B_KEY_DOWN

B_KEYBOARD_DEVICE

Purpose: Deliverable
Source: The input server.
Target: A keyboard device add-on.
Hook: `BInput::Control()`

This message is used by the input server to send a request to a keyboard device add-on.

Field	Type code	Description
"code"	B_INT32_TYPE	Operation code for the keyboard device control request being issued. One of: B_KEY_MAP_CHANGED B_KEY_LOCKS_CHANGED B_KEY_REPEAT_DELAY_CHANGED B_KEY_REPEAT_RATE_CHANGED
"device"	B_STRING_TYPE	The name of the device. If this field doesn't exist, the device is identified by "type" instead.
"type"	B_INT32_TYPE	The device type. If this field doesn't exist, the "device" field identifies the device.
"message"	B_MESSAGE_TYPE	The control message for the request. None of the BeOS standard messages use this field.

B_UNMAPPED_KEY_UP see B_KEY_DOWN

B_MEDIA_BUFFER_CREATED , B_MEDIA_BUFFER_DELETED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent to indicate that a media buffer has been created or deleted.

`B_MEDIA_BUFFER_CREATED`'s message has the following fields:

Field	Type Code	Description
"clone_info"	`B_RAW_TYPE`	An area_info structure describing the buffer's location in memory. This is an array, one entry per buffer created.

`B_MEDIA_BUFFER_DELETED` looks like this:

Field	Type Code	Description
"media_buffer_id"	`B_INT32_TYPE`	The buffer ID number of the buffer being deleted. This is an array, one entry per buffer deleted.

B_MEDIA_BUFFER_DELETED see B_MEDIA_BUFFER_CREATED

B_MEDIA_CONNECTION_BROKEN see B_MEDIA_CONNECTION_MADE

B_MEDIA_CONNECTION_MADE , B_MEDIA_CONNECTION_BROKEN

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent to indicate that a connection between media nodes has been made or broken.

B_MEDIA_FLAVORS_CHANGED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent by the media server to indicate that the flavors supported by a particular add-on have changed.

Field	Type Code	Description
"be:addon_id"	`B_INT32_TYPE`	The add-on ID of the add-on whose flavors have changed
"be:new_count"	`B_INT32_TYPE`	How many new flavors have been added.
"be:gone_count"	`B_INT32_TYPE`	How many flavors have been removed.

B_MEDIA_FORMAT_CHANGED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent by the media server to indicate that the format used on a particular connection has changed.

Field	Type Code	Description
"be:source"	`B_RAW_TYPE`	A media_source structure describing the source of the connection whose format changed.
"be:destination"	`B_RAW_TYPE`	A media_source structure describing the source of the connection whose format changed.
"be:format"	`B_RAW_TYPE`	A media_format structure describing the new format.

B_MEDIA_NODE_CREATED , B_MEDIA_NODE_DELETED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent to indicate that a node has been created or deleted in the media system.

Field	Type Code	Description
"media_node_id"	`B_INT32_TYPE`	The ID of the media_node that was created or deleted.

B_MEDIA_NODE_DELETED see B_MEDIA_NODE_CREATED

B_MEDIA_NODE_STOPPED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Indicates that a media node has stopped.

B_MEDIA_PARAMETER_CHANGED , B_MEDIA_NEW_PARAMETER_VALUE

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent by the media server to indicate that the value of a parameter has changed. `B_MEDIA_PARAMETER_CHANGED` only tells you which parameter changed (media_node and parameter ID). **`B_MEDIA_NEW_PARAMTER_VALUE`** also tells you which parameter changed, and what the new value is.

The `B_MEDIA_PARAMETER_CHANGED` message looks like this:

Field	Type Code	Description
"be:node"	`B_RAW_TYPE`	A media_node structure indicating which node's parameter web has changed.
"be:parameter"	`B_INT32_TYPE`	The ID number of the parameter whose value has changed.

The `B_MEDIA_NEW_PARAMETER_VALUE` message is:

Field	Type Code	Description
"node"	`B_RAW_TYPE`	A media_node structure indicating which node is reporting a changed parameter value.
"parameter"	`B_INT32_TYPE`	The parameter ID of the changed parameter.
"when"	`B_INT64_TYPE`	The performance time at which the change took effect.
"value"	`B_RAW_TYPE`	The parameter's new value.

B_MEDIA_WEB_CHANGED

Purpose: Deliverable
Source: The media server.
Target: Target specified to `BMediaRoster::StartWatching()`.

Sent by the media server to indicate that a particular node's BParameterWeb has.

Field	Type Code	Description
"node"	`B_RAW_TYPE`	A media_node structure indicating which node's parameter web has changed.

B_MEDIA_WILDCARD

Purpose: Constant only.
Source: Your application.
Target: The media server.

When calling `BMediaRoster::StartWatching()` or `BMediaRoster::StopWatching()`, you can use this constant to match all media messages. This message has no actual format.

(No Be-defined fields)

B_MINIMIZE

Source: The system or your app.
Target: The BWindow that's hidden/unhidden.
Hook: `BWindow::Minimize()`

Sent when the user double-clicks a window's title bar (to hide the window), or selects a window from the **DeskBar**'s window list (to unhide the window).

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70.
"minimize"	`B_BOOL_TYPE`	`true` if the window is being hidden; `false` if it's being unhidden.

B_MODIFIERS_CHANGED

Source: The system.
Target: The focus view's window.

Sent when the user presses or releases a modifier key.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"modifiers"	`B_INT32_TYPE`	The current modifier keys. See <x>
"be:old_modifiers"	`B_INT32_TYPE`	The previous modifier keys.
"states"	`B_UINT8_TYPE`	The state of all keys at the time of the event. See <x>.

B_MOUSE_DOWN

Source: The system.
Target: The BWindow of the view the mouse is pointing to.
Hook: `BView::MouseDown()`

Sent when the user presses a mouse button. This message is only sent if no other mouse button is already down.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"where"	`B_POINT_TYPE`	Mouse location in the view's coordinate system.
"modifiers"	`B_INT32_TYPE`	The modifier keys that were in effect at the time of the event.

"buttons"	`B_INT32_TYPE`	The mouse button that was pressed, one of:
		`B_PRIMARY_MOUSE_BUTTON`
		`B_SECONDARY_MOUSE_BUTTON`
		`B_TERTIARY_MOUSE_BUTTON`

"clicks"	`B_INT32_TYPE`	1 for a single-click, 2 for double-click, 3 for triple-click, and so on. The counter is reset if the time between clicks exceeds the "Double-

		click speed" set by the user in the **Mouse** preferences. Note that the counter is *not* reset if the mouse moves between clicks.

B_MOUSE_MOVED

Source: The system.
Target: The BWindow of the view the mouse is pointing to.
Hook: `BView::MouseMoved()`

Sent when the user moves the mouse.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"where"	`B_POINT_TYPE`	The mouse's new location in window coordinates.

"buttons"	`B_INT32_TYPE`	The mouse buttons that are down. Zero or more of:
		`B_PRIMARY_MOUSE_BUTTON`
		`B_SECONDARY_MOUSE_BUTTON`
		`B_TERTIARY_MOUSE_BUTTON`

B_MOUSE_UP

Source: The system.
Target: The BWindow of the view the mouse is pointing to.
Hook: `BView::MouseUp()`

Sent when the user releases a mouse button. It's only sent if no other mouse button remains down.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"where"	`B_POINT_TYPE`	Mouse location in the view's coordinate system.
"modifiers"	`B_INT32_TYPE`	The modifier keys that were in effect at the time of the event. See <x> for a list of modifier values.

B_MOUSE_WHEEL_CHANGED

Source: The system.
Target: The BWindow of the view the mouse is pointing to.

Sent when the user moves the mouse wheel (on mice that have them).

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"be:wheel_delta_x"	`B_FLOAT_TYPE`	How much the Y value of the wheel has changed.
"be:wheel_delta_y"	`B_FLOAT_TYPE`	How much the Y value of the wheel has changed.

The standard mouse driver that comes with BeOS only supports a Y-oriented wheel.

B_NODE_MONITOR

Source: The Node Monitor.
Target: App-defined.

Sent when a monitored file changes. All Node Monitor messages contain an **`int32`** "opcode" field that tells you what happened (a file was removed, an attribute changed, etc.). The format of the rest of the message depends on the opcode. The formats are given under individual entries in this appendix, listed by opcode value:

- **`B_ENTRY_CREATED`**
- **`B_ENTRY_REMOVED`**
- **`B_ENTRY_MOVED`**
- **`B_STAT_CHANGED`**
- **`B_ATTR_CHANGED`**
- **`B_DEVICE_MOUNTED`**
- **`B_DEVICE_UNMOUNTED`**

These are discussed in the section "Node Monitor Messages".

B_OBSERVER_NOTICE_CHANGE

Source: The system.
Target: Application-defined target.

Sent by BHandlers to all targets that have been registered with the BHandler's **`StartWatching()`** or **`StartWatchingAll()`** function.

Field	Type code	Description
"be:observe_change_what"	**`B_INT32_TYPE`**	The "what" code of the message being broadcast.
"be:observe_orig_what"	**`B_INT32_TYPE`**	The original "what" code of the message, before it was altered by the broadcasting mechanism.

The message may have other fields, depending on what the broadcast message is. Messages are sent to targets in response to the **`BHandler::SendNotices()`** function. The logic is:

- Take the message to broadcast and place its what code into the "be:observe_change_orig_what" field.

- Add the "be:observe_change_what" field, which is set to the what code specified by the call to `SendNotices()`.
- Set the message's what code to `B_OBSERVER_NOTICE_CHANGE`.

The resulting message is then sent to all interested parties.

See also: `BHandler::SendNotices()`, `BHandler::StartWatching()`

B_OPEN_IN_WORKSPACE

Source: The system.
Target: BApplication.

Sent to an application when it's first launched to tell it to open in a specific workspace. The message will be handled during the construction of the BApplication object.

Field	Type code	Description
"be:workspace"	`B_INT32_TYPE`	Workspace number into which the application should open.

B_POINTING_DEVICE

Purpose: Deliverable
Source: The input server.
Target: A pointing device add-on.
Hook: `BInput::Control()`

This message is used by the input server to send a request to a pointing device add-on.

Field	Type code	Description

"code"	`B_INT32_TYPE`	Operation code for the pointing device control request being issued. One of:
		`B_MOUSE_TYPE_CHANGED`
		`B_MOUSE_MAP_CHANGED`
		`B_MOUSE_SPEED_CHANGED`
		`B_CLICK_SPEED_CHANGED`
		`B_MOUSE_ACCELERATION_CHANGED`

"device"	`B_STRING_TYPE`	The name of the device. If this field doesn't exist, the device is identified by "type" instead.
"type"	`B_INT32_TYPE`	The device type. If this field doesn't exist, the "device" field identifies the device.
"message"	`B_MESSAGE_TYPE`	The control message for the request. None of the BeOS standard messages use this field.

B_PRINTER_CHANGED

Source: The Print Server.
Target: Everyone.

Sent whenever the user changes printers. Applications that support printing should watch for this message, and if they receive it, they should suggest that the user check their page setup the next time they choose to print.

B_PULSE

Source: The system.
Target: `be_app` or BWindow object.
Hook: `BApplication::Pulse()` and `BView::Pulse()`

Sent at regular intervals, but with no particular intent. You can implement `Pulse()` to do whatever you want. The message is to the BWindow only if a BView within the window declares `B_PULSE_NEEDED` in its constructor flags.

B_QUERY_UPDATE

Source: The system.
Target: App-defined.

Sent when the results of a live query change: If a new file meets the query requirements, `B_ENTRY_CREATED` is sent. If a file that previously passed the query requirements no longer does, `B_ENTRY_REMOVED` is sent.

B_QUIT_REQUESTED

Source: The system or your app.
Target: `be_app`, BWindow closed by the user, or other BLooper object.
Hook: `BLooper::QuitRequested()`.

Automatically sent **(a)** to `be_app` when the user types **Command**+**q**, and **(b)** to a BWindow when the user clicks the window's close box. Applications can also manufacture and send the message themselves. A looper that receives this message is expected to quit, or at least consider quitting.

Field	Type code	Description
"shortcut"	`B_BOOL_TYPE`	true if the user typed **Command**+**q**.

B_READY_TO_RUN

Source: The system.
Target: `be_app`
Hook: `BApplication::ReadyToRun()`

Sent when an application has finished configuring itself and is ready to start running.

(No Be-defined fields)

B_REFS_RECEIVED

Source: The system or your app.
Target: `be_app`, or other app-defined target.
Hook: `BApplication::RefsReceived()`

Automatically sent to `be_app` when **(a)** the user double-clicks a file that has a type that's supported by the app, and **(b)** when the user confirms some files (or directories) in an **Open File** panel (the target is `be_app` by default; it can be changed in `BFilePanel::SetTarget()`). You can also create, stuff, and send a `B_REFS_RECEIVED` message yourself. When it receives this message, an app is expected to open the files that the message refers to.

Field	Type code	Description
"refs" [*i*]	`B_REF_TYPE`	`entry_ref` items, one for each file or directory.

A BStatusBar object can be controlled synchronously by calling its `Reset()` and `Update()` functions. It can also be controlled asynchronously by sending it messages corresponding to the two functions; the object calls the function when it receives the message. Each message contains fields for the arguments passed to the function.

B_RELEASE_OVERLAY_LOCK see B_ACQUIRE_OVERLAY_LOCK

B_RESET_STATUS_BAR

Source: Your app.
Hook: `BStatusBar::Reset()`
Target: The BStatusBar you're resetting.

You construct and send this message to a BStatusBar object to tell it to (asynchronously) reset itself. The message also lets you reset the object's labels. To send the message, invoke BWindow's `PostMessage()` naming the target BStatusBar as the handler:

```
statusBar->Window()->PostMessage(theMessage, statusBar);
```

Field	Type code	Description
"label"	`B_STRING_TYPE`	The object's new label (`NULL`-terminated).
"trailing_label"	`B_STRING_TYPE`	The object's new trailing label (`NULL`-terminated)..

B_SCREEN_CHANGED

Source: The system.
Target: Every BWindow in the screen that changed (even hidden windows).
Hook: `BWindow::ScreenChanged()`

Sent when the screen's dimensions or color space changes—because the user played with the **Screen** preferences app, for example.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"frame"	B_RECT_TYPE	The screen's dimensions.
"mode"	B_INT32_TYPE	The screen's color space: B_CMAP8, B_RGB15, B_RGB15, or BRGB32.

B_SILENT_RELAUNCH

Source: The system.
Target: be_app

Sent to a single-launch application when it's activated by being launched (for example, if the user double-clicks its icon in Tracker).

(No Be-defined fields)

B_SOME_APP_ACTIVATED , B_SOME_APP_LAUNCHED , B_SOME_APP_QUIT

Source: The Roster Monitor.
Target: App-defined.

Sent as apps are launched, activated, or quit. You get these messages by invoking BRoster::StartWatching() passing a one or more of B_REQUEST_ACTIVATED, B_REQUEST_LAUNCHED, and B_REQUEST_QUIT.

Field	Type code	Description
"mime_sig"	B_STRING_TYPE	The ap signature.
"team"	B_INT32_TYPE	The app's team id.
"thread"	B_INT32_TYPE	The id of the app's main thread.
"flags"	B_INT32_TYPE	The app's app flags (B_SINGLE_LAUNCH, B_BACKGROUND_APP, etc).
"ref"	B_REF_TYPE	The entry_ref of the app's executable.

B_UPDATE_STATUS_BAR

Source: Your app.
Hook: BStatusBar::Update()
Target: The BStatusBar you're updating.

You construct and send this message to a BStatusBar object to tell it to (asynchronously) update its progress. To send the message, invoke BWindow's PostMessage() naming the target BStatusBar as the handler:

```
statusBar->Window()->PostMessage(theMessage, statusBar);
```

Field	Type code	Description
"delta"	B_FLOAT_TYPE	An increment to the object's current value.
"text"	B_STRING_TYPE	The object's new text (NULL-terminated).
"trailing_text"	B_STRING_TYPE	The object's new trailing text (NULL-terminated)..

B_VALUE_CHANGED

Source: The system.
Target: The manipulated scrollbar's BWindow.
Hook: BScrollBar::ValueChanged()

Sent when the user plays with a scrollbar.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"value"	B_INT32_TYPE	The scrollbar's new value.

B_VIEW_MOVED

Source: The system.
Target: The moved view's BWindow.
Hook: BView::FrameMoved()

Sent when a view's origin (left top corner) changes relative to the origin of its parent. The message isn't sent if the view doesn't have the B_FRAME_EVENTS flag set.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"where"	B_POINT_TYPE	The view's new origin in the coordinate system of its parent.

B_VIEW_RESIZED

Source: The system.
Target: The resized view's BWindow.
Hook: BView::FrameResized()

Sent when the size of the view's frame changes. The message isn't sent if the view doesn't have the B_FRAME_EVENTS flag set.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"width"	B_INT32_TYPE	The view's new width.

"height"	B_INT32_TYPE	The view's new height.
"where"	B_POINT_TYPE	The view's new origin expressed in the coordinate system of its parent. This field is only included if the view actually moved while being resized, and can always be ignored: If the view *did* move, you'll hear about it in a separate B_VIEW_MOVED BMessage.

B_WINDOW_ACTIVATED

Source: The system.
Target: The activated/deactivated BWindow.
Hook: BWindow::WindowActivated() and BView::WindowActivated()

Sent just after a window is activated or deactivated. Note that the BWindow invokes WindowActivated() on each of its views.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"active"	B_BOOL_TYPE	true if the window is now active; false if not.

B_WINDOW_MOVE_BY

Purpose: Deliverable.
Source: Your application.
Target: The BWindow to be moved.

You can send this message to a window to resize it by the specified deltas.

Field	Type code	Description
"data"	B_POINT_TYPE	The amount by which to move the window's X and Y coordinates.

B_WINDOW_MOVE_TO

Purpose: Deliverable.
Source: Your application.
Target: The BWindow to be moved.

You can send this message to a window to resize it to the specified size.

Field	Type code	Description
"data"	B_POINT_TYPE	The width and height (in X and Y) to resize the window to.

B_WINDOW_MOVED

Source: The system.
Target: The BWindow that moved.

Hook: `BWindow::FrameMoved()`

Sent when a window's origin (left top corner) changes within the screen coordinate system.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"where"	`B_POINT_TYPE`	The window's new origin in screen coordinates.

B_WINDOW_RESIZED

Source: The system.
Target: The resized BWindow.
Hook: `BWindow::FrameResized()`

Sent when the size of the window's frame changes. Note that the "width" and "height" fields measure the window's content area—they don't include the window border or window tab.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"width"	`B_INT32_TYPE`	The width of the window's content area.
"height"	`B_INT32_TYPE`	The height of the window's content area.

B_WORKSPACE_ACTIVATED

Source: The system.
Target: Every BWindow in the activated and deactivated workspaces.
Hook: `BWindow::WorkspaceActivated()`

Sent when the active workspace changes.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"workspace"	`B_INT32_TYPE`	The index of the window's workspace.
"active"	`B_BOOL_TYPE`	**`true`** if the workspace is now active; **`false`** if not.

B_WORKSPACES_CHANGED

Source: The system.
Target: The BWindow whose set of workspaces changed.
Hook: `BWindow::WorkspacesChanged()`

Sent when there's a change to the set of workspaces in which a window can appear.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70

"old"	B_INT32_TYPE	The window's old workspace set, given as a vector of workspace indices.
"new"	B_INT32_TYPE	The window's new workspace set, given as a vector of workspace indices.

B_ZOOM

Source: The system.
Target: The BWindow that was zoomed.
Hook: BWindow::Zoom()

Sent when the user clicks a window's zoom button.

The message has just one data field:

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Messages Table of Contents

- System Messages Introduction
- General Messages
- Data Containers
- File Panel Messages
- Node Monitor Messages
- Reply Messages
- Scripting Messages

The Be Book Messages Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Messages: Master Index

A	General Messages

A

B_ABOUT_REQUESTED	General Messages
B_ACQUIRE_OVERLAY_LOCK	General Messages
B_APP_ACTIVATED	General Messages
B_ARCHIVED_OBJECT	General Messages
B_ARGV_RECEIVED	General Messages
B_ATTR_CHANGED	Node Monitor Messages

C

B_CANCEL	File Panel Messages
B_CANCEL	General Messages
B_CLIPBOARD_CHANGED	General Messages
B_CONTROL_INVOKED	General Messages
B_COUNT_PROPERTIES	Scripting Messages
B_CREATE_PROPERTY	Scripting Messages

D

Data Containers	Data Containers

Data Containers	Data Containers
B_DELETE_PROPERTY	Scripting Messages
B_DEVICE_MOUNTED	Node Monitor Messages
B_DEVICE_UNMOUNTED	Node Monitor Messages

E

B_ENTRY_CREATED	Node Monitor Messages
B_ENTRY_MOVED	Node Monitor Messages
B_ENTRY_REMOVED	Node Monitor Messages

F

File Panel Messages	File Panel Messages
File Panel Messages	File Panel Messages

G

General Messages	General Messages
General Messages	General Messages
B_GET_PROPERTY	Scripting Messages
B_GET_SUPPORTED_SUITES	Scripting Messages

I

B_INPUT_DEVICES_CHANGED	General Messages
B_INPUT_METHOD_EVENT	General Messages

K

B_KEY_DOWN	General Messages
B_KEY_UP	General Messages
B_KEYBOARD_DEVICE	General Messages

M

B_MINIMIZE	General Messages
B_MEDIA_BUFFER_CREATED	General Messages
B_MEDIA_BUFFER_DELETED	General Messages
B_MEDIA_CONNECTION_BROKEN	General Messages
B_MEDIA_CONNECTION_MADE	General Messages
B_MEDIA_FLAVORS_CHANGED	General Messages
B_MEDIA_FORMAT_CHANGED	General Messages
B_MEDIA_NEW_PARAMETER_VALUE	General Messages
B_MEDIA_NODE_CREATED	General Messages
B_MEDIA_NODE_DELETED	General Messages
B_MEDIA_NODE_STOPPED	General Messages
B_MEDIA_PARAMETER_CHANGED	General Messages
B_MEDIA_WEB_CHANGED	General Messages
B_MEDIA_WILDCARD	General Messages
B_MESSAGE_NOT_UNDERSTOOD	Reply Messages
B_MIME_DATA	Data Containers
B_MODIFIERS_CHANGED	General Messages

B_MOUSE_DOWN	General Messages
B_MOUSE_MOVED	General Messages
B_MOUSE_UP	General Messages
B_MOUSE_WHEEL_CHANGED	General Messages

N

B_NO_REPLY	Reply Messages
Node Monitor Messages	Node Monitor Messages
Node Monitor Messages	Node Monitor Messages
B_NODE_MONITOR	General Messages

O

B_OBSERVER_NOTICE_CHANGE	General Messages
B_OPEN_IN_WORKSPACE	General Messages

P

B_PULSE	General Messages
B_POINTING_DEVICE	General Messages
B_PRINTER_CHANGED	General Messages

Q

B_QUERY_UPDATE	General Messages
B_QUIT_REQUESTED	General Messages

R

B_REPLY	Reply Messages
Reading a Message Description	System Messages Introduction
B_READY_TO_RUN	General Messages
B_REFS_RECEIVED	General Messages
B_RELEASE_OVERLAY_LOCK	General Messages
Reply Messages	Reply Messages
Reply Messages	Reply Messages
B_RESET_STATUS_BAR	General Messages

S

B_SAVE_REQUESTED	File Panel Messages
B_SCREEN_CHANGED	General Messages
Scripting Messages	Scripting Messages
Scripting Messages	Scripting Messages
B_SET_PROPERTY	Scripting Messages
B_SILENT_RELAUNCH	General Messages
B_SIMPLE_DATA	Data Containers
B_SOME_APP_ACTIVATED	General Messages
B_SOME_APP_LAUNCHED	General Messages
B_SOME_APP_QUIT	General Messages
B_STAT_CHANGED	Node Monitor Messages
System Messages Introduction	System Messages Introduction
System Messages Introduction	System Messages Introduction

U

B_UNMAPPED_KEY_DOWN	General Messages
B_UNMAPPED_KEY_UP	General Messages
B_UPDATE_STATUS_BAR	General Messages

V

B_VALUE_CHANGED	General Messages
B_VIEW_MOVED	General Messages
B_VIEW_RESIZED	General Messages

W

B_WINDOW_ACTIVATED	General Messages
B_WINDOW_MOVE_BY	General Messages
B_WINDOW_MOVE_TO	General Messages
B_WINDOW_MOVED	General Messages
B_WINDOW_RESIZED	General Messages
B_WORKSPACE_ACTIVATED	General Messages
B_WORKSPACES_CHANGED	General Messages

Z

B_ZOOM	General Messages

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Node Monitor Messages

The Node Monitor is a mechanism that lets you watch for changes to a particular file or directory (or "node"). You ask the Node Monitor to start or stop watching a given node by calling the `watch_node()` function; you can stop all monitoring through the `stop_watching()` function.

When you call `watch_node()`, you tell the Node Monitor which aspects of the node you want to track—changes to its name, to its size, its attributes, and so on. Each of these "trackable" elements corresponds to a particular type of message (identified by the message's "opcode" field) that's sent back to your application when that element actually changes (when the file is renamed, changes size, gains an attribute, and so on).

B_ENTRY_CREATED

Field	Type code	Description
"opcode"	`B_INT32_TYPE`	`B_ENTRY_CREATED` indicates that a new entry was created.
"name"	`B_STRING_TYPE`	The name of the new entry.
"directory"	`B_INT64_TYPE`	The **`ino_t`** (node) number for the directory in which the entry was created.
"device"	`B_INT32_TYPE`	The **`dev_t`** number of the device on which the new entry resides.

"node"	B_INT64_TYPE	The **ino_t** number of the new entry itself. (More accurately, it identifies the node that corresponds to the entry.)

B_ENTRY_REMOVED

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_ENTRY_REMOVED indicates that an entry was removed.
"directory"	**B_INT64_TYPE**	The **ino_t** (node) number of the directory from which the entry was removed.
"device"	**B_INT32_TYPE**	The **dev_t** number of the device that the removed node used to live on.
"node"	**B_INT64_TYPE**	The **ino_t** number of the node that was removed.

B_ENTRY_MOVED

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ENTRY_MOVED indicates that an existing entry moved from one directory to another.
"name"	B_STRING_TYPE	The name of the entry that moved.
"from directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory from that the node was

		removed from.
"to directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory that the node was added to.
"device"	B_INT32_TYPE	The **dev_t** number of the device that the moved node entry lives on. (You can't move a file between devices, so this value will be apply to the file's old and new locations.)
"node"	B_INT64_TYPE	The **ino_t** number of the node that was removed.

B_STAT_CHANGED

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_STAT_CHANGED indicates that some statistic of a node (as recorded in its **stat** structure) changed.
"node"	**B_INT64_TYPE**	The **ino_t** number of the node.
"device"	**B_INT32_TYPE**	The **dev_t** number of the node's device.

B_ATTR_CHANGED

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_ATTR_CHANGED indicates that some attribute of a node changed.
"node"	**B_INT64_TYPE**	The **ino_t** number of the node.

"device"	**B_INT32_TYPE**	The **dev_t** number of the node's device.

B_DEVICE_MOUNTED

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_DEVICE_MOUNTED indicates that a new device (or file system volume) has been mounted.
"new device"	**B_INT32_TYPE**	The **dev_t** number of the newly-mounted device.
"device"	**B_INT32_TYPE**	The **dev_t** number of the device that holds the directory of the new device's mount point.
"directory"	**B_INT64_TYPE**	The **ino_t** (node) number of the directory that acts as the new device's mount point.

B_DEVICE_UNMOUNTED

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_DEVICE_UNMOUNTED indicates that a device has been unmounted.
"new device"	**B_INT32_TYPE**	The **dev_t** number of the unmounted device.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Node Monitor

Declared in: be/storage/NodeMonitor.h
Library: libbe.so
Summary: *more...*

The Node Monitor is a service that lets you ask to be notified of certain file system changes. You can ask to be told when a change is made to...

- The contents of a specific directory.
- The name of a specific entry.
- Any stat field of a specific entry.
- Any attribute of a specific entry.

You can also ask to be notified when...

- Volumes are mounted and unmounted.

Volume monitoring is also provided by the BVolumeRoster class: BVolumeRoster can talk to the Node Monitor for you. The BVolumeRoster volume-watching API is more humane than that which you'll find here.

When something interesting happens, the Node Monitor lets you know by sending a BMessage to the target of your choice.

Node Monitor Functions

There are two Node Monitor functions, **watch_node()** and **stop_watching()**. The names are a wee bit misleading, so before we go on to the full technical descriptions, let's nip some buds:

- **watch_node()** tells the Node Monitor to start *or stop* watching a *specific* node, or to watch for volumes being mounted and unmounted. Memorize the emphasized words.
- **stop_watching()** tells the Node Monitor to stop sending notifications to a particular target.

---

watch_node()

```
status_t watch_node(const node_ref *nref,
    uint32 flags,
    BMessenger messenger)
status_t watch_node(const node_ref *nref,
    uint32 flags,
    const BHandler *handler,
    const BLooper *looper = NULL)
```

watch_node() tells the Node Monitor to...

- ...start paying attention to the node specified by the **node_ref** argument. If you're watching for volumes (only), *nref* can be **NULL**. The easiest way to get a **node_ref** is to invoke **BStatable::GetNodeRef()** on any BEntry or BNode object.
- The *flags* argument lists the changes that you want the Monitor to pay attention to. See below for details.
- The target of the change notification messages is specified either as a BMessenger, or as a BHandler*/BLooper* pair. (The target specification follows the **BInvoker::SetTarget()** protocol; see the BInvoker class for details.) The notification shows up as a BMessage in the target's **MessageReceived()** function.

You can't tell the Node Monitor to send its notifications to another application. Currently, the BMessenger that you specify must identify a target in the caller's team.

Jumping ahead a bit, here's a sample function that tells the Node Monitor to watch for name and attribute changes to a given entry. The Monitor's notifications will be sent to the application's main loop:

```
status_t WatchThis(BEntry *entry)
{
   node_ref nref;
   entry->GetNodeRef(&nref);
   return (watch_node(&nref,
```

```
            B_WATCH_NAME | B_WATCH_ATTR,
            be_app_messenger));
}
```

Monitor Flags

`watch_node()`'s *flags* argument is a combination of the following

- **`B_WATCH_NAME`** watches for name changes. This includes moving the node to a different directory, or removing the node altogether.
- **`B_WATCH_STAT`** watches for *any* change to the node's **`stat`** structure. This includes changes to the size, modification date, owner, and so on. See "The stat Structure" on page155 in the BStatable class for a description of what's in the **`stat`** structure.
- **`B_WATCH_ATTR`** watches for changes to *any* of the node's attributes. This includes adding and removing attributes.
- **`B_WATCH_DIRECTORY`** only applies to nodes that are directories. The flag tells the Monitor to watch for changes (new entries, entry deletions, entries being renamed) to the directory. (You can apply the other flags to a directory, as well). It's not an error to set **`B_WATCH_DIRECTORY`** on a node that isn't a directory—but it doesn't *do* anything for you.
- **`B_WATCH_ALL`**. This is a convenience that combines all the above.
- **`B_WATCH_MOUNT`** watches for volumes being mounted and unmounted. As mentioned above, the *nref* argument isn't needed (it can be **`NULL`**) if all you're doing is watching volumes. **`B_WATCH_MOUNT`** *isn't* included in **`B_WATCH_ALL`**.

There's one other constant, which lives in a class by itself:

- **`B_STOP_WATCHING`** tells the Node Monitor to stop watching the *nref* argument.

You can't combine **`B_STOP_WATCHING`** with any of the others in an attempt to stop watching a specific category of changes. For example, if you call...

```
watch_node(&nref, B_WATCH_STAT, be_app_messenger);
watch_node(&nref, B_WATCH_ATTR, be_app_messenger);
```

...and then call...

```
watch_node(&nref, B_STOP_WATCHING, be_app_messenger);
```

...both of the previous Monitor calls are stopped.

`B_STOP_WATCHING` does *not* apply to volume watching. The only way to stop monitoring volume un/mounts is to call **`stop_watching()`**.

Combining Flags and the 4096 Limit

If you can, you should combine as many flags as you're going to need in single calls to **watch_node()**. Recall the example used above:

```
watch_node(&nref,
      B_WATCH_NAME | B_WATCH_ATTR,
      be_app_messenger);
```

This is better than making separate **watch_node()** calls (one to pass **B_WATCH_NAME** and another to pass **B_WATCH_ATTR**)—not only because the single call is naturally more efficient than two, but also because **the Node Monitor can only monitor 4096 nodes per team at a time**. Every call to **watch_node()** consumes a Node Monitor slot, even if you're *already* monitoring the requested node.

If you want to watch all aspects of a node, just pass **B_WATCH_ALL** to every **watch_node()** call. This will consume only a single Node Monitor slot.

Notification Messages

A BMessage notification sent by the Node Monitor looks like this:

- The `what` value is **B_NODE_MONITOR**.
- The field named "opcode" is an **int32** constant that tells you what happened.
- Additional fields give you information (device, node, name, and so on) about the node (or volume) that it happened to.

The "opcode" constants and additional fields are described in "Opcode Constants." In general, the opcodes correspond to the flags that you passed to **watch_node()**; however, this correspondence isn't always one-to-one.

There are seven opcode constants:

B_ENTRY_CREATED
B_ENTRY_REMOVED
B_ENTRY_MOVED
B_STAT_CHANGED
B_ATTR_CHANGED
B_DEVICE_MOUNTED
B_DEVICE_UNMOUNTED

RETURN CODES

- **B_OK**. The Node Monitor is off and running.
- **B_BAD_VALUE**. Bad *nref* argument (not applicable to mount-only watches), or poorly formed target.
- **B_NO_MEMORY**. Couldn't allocate resources, or out of Node Monitor slots.

- B_ERROR. Some cases of bad *nref* arguments erroneously return B_ERROR. This will be fixed.

---

stop_watching()

status_t **stop_watching(**BMessenger *messenger***)**
status_t **stop_watching(**const BHandler ***handler,** const BLooper ***looper** = *NULL***)**

Tells the Node Monitor to stop sending notifications to the target described by the arguments. All the Node Monitor "slots" that were allocated to the target are freed. Keep in mind that are only 4096 slots for the entire system.

RETURN CODES

- B_OK. The target is now out of the Node Monitor loop.
- B_BAD_VALUE. Badly formed target description.

Opcode Constants

The following sections describe the "opcode" constants; these are the values that appear in the "opcode" field of the BMessages that are generated by the Node Monitor. Note that in these descriptions, the use of the terms "entry" and "node" is sometimes blurred.

B_ENTRY_CREATED

- A completely new entry was created in a monitored directory. (This doesn't include entries that are moved into this directory from some other directory—see **B_ENTRY_MOVED**.)

You get this notification if you applied **B_WATCH_DIRECTORY** to the directory in which the entry was created. The message's fields are:

Field	Type code	Description
"opcode"	**B_INT32_TYPE**	B_ENTRY_CREATED
"name"	**B_STRING_TYPE**	The name of the new entry.
"directory"	**B_INT64_TYPE**	The **ino_t** (node) number for the directory in which the entry was created.
"device"	**B_INT32_TYPE**	The **dev_t** number of the device on which the new entry resides.

Parsing and Tricks

In your code, you would parse a B_ENTRY_CREATED message like this:

```
void MyTarget::MessageReceived(BMessage *msg)
{
   int32 opcode;
   dev_t device;
   ino_t directory;
   ino_t node;
   const char *name;

   if (msg->what == B_NODE_MONITOR) {
      if (msg->FindInt32("opcode", &opcode) == B_OK) {
         switch (opcode) {
            case B_ENTRY_CREATED:
               msg->FindInt32("device", &device);
               msg->FindInt64("directory", &directory);
               msg->FindInt64("node", &node);
               msg->FindString("name", &name);
               break;
            ...
```

So, what do you do with these fields?

Create an `entry_ref` to the entry.

The "device", "directory", and "name" fields can be used to create an **`entry_ref`** to the new entry:

```
entry_ref ref;
const char *name;
...
msg->FindInt32("device", &ref.device);
msg->FindInt64("directory", &ref.directory);
msg->FindString("name", &name);
ref.set_name(name);
```

Create a `node_ref` to the entry.

If you want to start Node Monitoring the new entry (or, more accurately, the node of the new entry), you stuff "device" and "directory" into a **`node_ref`**:

```
node_ref nref;
status_t err;

...
msg->FindInt32("device", &nref.device);
msg->FindInt64("node", &nref.node);

err = watch_node(&nref, B_WATCH_ALL, be_app_messenger);
```

Create a `node_ref` to the entry's parent.

Note that the "directory" field is a node number. By combining this number with the "device" field, you can create a **`node_ref`** that points to the entry's parent. From there, you're a **`SetTo()`** away from a BDirectory object:

```
node_ref nref;
BDirectory dir;
status_t err;

...
msg->FindInt32("device", &nref.device);
msg->FindInt64("directory", &nref.node);
err = dir.SetTo(&nref);
```

B_ENTRY_REMOVED

- A node was removed (deleted) from a directory.

You get this if you applied **B_WATCH_NAME** on the node itself, or **B_WATCH_DIRECTORY** on the directory that the node lived in. The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ENTRY_REMOVED
"directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory from which the entry was removed.
"device"	B_INT32_TYPE	The **dev_t** number of the device that the removed node used to live on.
"node"	B_INT64_TYPE	The **ino_t** number of the node that was removed.

Since this message is telling you that the node was removed, the "node" value will be invalid. The node number *can* be useful (and sometimes necessary) for comparison with cached node numbers (as demonstrated below).

Parsing the message is the same as for **B_ENTRY_CREATED**, but without the "name" field. See "Parsing and Tricks," above.

Note that the **B_ENTRY_REMOVED** message is sent as soon as the node's entry is "unlinked" from its directory. The node itself may linger for while after that. Follow this logic:

- When a file (regardless of flavor) is removed, the entry for that file is *immediately* removed ("unlinked") from the file hierarchy, and the Node Monitor message is *immediately* sent—even if you have an object that has opened the file's node.
- The node isn't actually destroyed until the last open object (to that node) is destroyed. (In POSIX speak, the node is destroyed when the last file descriptor to the node is closed.)
- Until the node is destroyed, the open objects (file descriptors) can still access the node's data.

You can take advantage of this to warn a user that a file is going to go away, or to make a backup, or whatever. For example, let's say you have an application that lets the user open files; each time a file is opened, your **OpenFile()** function creates a BFile object and starts the Node Monitor running:

```
status_t YourApp::OpenFile(const char *pathname)
{
   BFile *file;
   node_ref nref;
   status_t err;

   file = new BFile(pathname, B_READ_WRITE);
   if ((err=file->InitCheck()) != B_OK)
      return err;

   file->GetNodeRef(&nref);
```

```
   err = watch_node(&nref, B_WATCH_NAME, be_app_messenger);

   if (err != B_OK) {
      delete file;
      return err;
   }

   /* We've got the file and we're monitoring it; now we cache
   * the BFile by adding it to a BList (data member).
   * function. There's a race condition between the
   * watch_node() call above and the following AddItem().
   */
   return ((FileList->AddItem((void *)file)) ? B_OK : B_ERROR);

}
```

Now we receive a Node Monitor message telling us the node has been removed. We stuff the "device" and "node" fields into a **node_ref** and pass them to a (fictitious) **AlertUser()** function:

```
void YourApp::MessageReceived(BMessage *msg)
{
   int32 opcode;
   node_ref nref;

   if (msg->what == B_NODE_MONITOR) {
      if (msg->FindInt32("opcode", &opcode) == B_OK) {
         switch (opcode) {
            case B_ENTRY_REMOVED:
               msg->FindInt32("device", &nref.device);
               msg->FindInt64("node", &nref.node);
               GoodbyeFile(nref);
   ...
}
```

The implementation of **GoodbyeFile()** (which we won't show here) would walk down the BFile list looking for a **node_ref** that matches the argument:

```
void YourApp::GoodbyeFile(node_ref nref)
{
   BFile *filePtr;
   int32 ktr = 0;
   node_ref cref;

   while ((*filePtr = (BFile *)FileList->ItemAt(ktr++))) {
      filePtr->GetNodeRef(&cref);
      if (nref == cref) {
         /* We found it. Now we do whatever
```

```
            * we need to do.
            */
        }
    }
}
```

If a match is found, your app could then do whatever it needs to do. Remember—the node's data is still valid until your BFile is destroyed or re-initialized.

B_ENTRY_MOVED

- A node was moved from one directory to a different directory.

You get this if you applied **B_WATCH_NAME** on the node itself, or **B_WATCH_DIRECTORY** on *either* of the directories. The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ENTRY_MOVED
"name"	B_STRING_TYPE	The name of the entry that moved.
"from directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory from that the node was removed from.
"to directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory that the node was added to.
"device"	B_INT32_TYPE	The **dev_t** number of the device that the moved node entry lives on. (You can't move a file between devices, so this value will be apply to the file's old and new locations.)
"node"	B_INT64_TYPE	The **ino_t** number of the node that moved.

Moving a node *does not* change its **ino_t** number.

Parsing the message is much the same as for **B_ENTRY_CREATED**, modulo the directory field changes. See "Parsing and Tricks."

Moving a node doesn't affect the objects that hold the node open. They (the objects) can continue to read and write data from the node.

B_STAT_CHANGED

- A field in the node's **stat** structure changed (this doesn't include the **stat** structure disappearing because the node was deleted).

You get this if you applied **B_WATCH_STAT** on the node itself. The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_STAT_CHANGED

"node"	B_INT64_TYPE	The **ino_t** number of the node.
"device"	B_INT32_TYPE	The **dev_t** number of the node's device.

The **stat** structure is described in "The stat Structure" on page155 in the BStatable class. The fields that you can change are:

- Owner (st_uid), group (st_gid), and permissions (low four bytes of st_mode).
- Creation (st_ctime), modification (st_mtime), and access times (st_atime; currently unused).
- The size of the node's data (st_size). The measurement doesn't include attributes.

A couple of important points:

- The B_STAT_CHANGED message doesn't give you enough information to construct an object from which you can get a **stat** structure. In other words, you can't play the same games that were described in "Parsing and Tricks."
- The message also doesn't tell you which **stat** field changed.

In most uses of the B_STAT_CHANGED message, you have to cache the objects that you're monitoring so you can compare their node_refs to the message fields (an example of this is given in B_ENTRY_REMOVED). Furthermore, you may want to cache the objects' **stat** structures so you can figure out which field changed.

B_ATTR_CHANGED

- An attribute of the node changed.

You get this if you applied **B_WATCH_ATTR** on the node itself. The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ATTR_CHANGED
"node"	B_INT64_TYPE	The **ino_t** number of the node.
"device"	B_INT32_TYPE	The **dev_t** number of the node's device.

Attributes are key/value pairs that can be "attached" to any file (regardless of flavor). They're described in the BNode class.

As with B_STAT_CHANGED messages, you may not be able to use the B_ATTR_CHANGED information directly. Instead, you have to cache references to the (BNode) objects that you're monitoring so you can compare their node_refs to the message fields (an example of this is given in B_ENTRY_REMOVED).

B_DEVICE_MOUNTED

- A file system device (in other words, a *volume*) was mounted.

You get this if you passed **B_WATCH_MOUNT** to watch_node(). The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_DEVICE_MOUNTED
"new device"	B_INT32_TYPE	The **dev_t** number of the newly-mounted device.
"device"	B_INT32_TYPE	The **dev_t** number of the device that holds the directory of the new device's mount point.
"directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory that acts as the new device's mount point.

Obviously, there's no node involved, here, so the first argument to the `watch_node()` call can be **NULL**:

```
watch_node(NULL, B_WATCH_MOUNT, be_app_messenger);
```

Unlike with the other "watch flags," the only way to stop the mount-watching is to call stop_watching().

B_DEVICE_UNMOUNTED

- A file system device (in other words, a *volume*) was unmounted.

You get this if you passed **B_WATCH_MOUNT** to watch_node(). The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_DEVICE_UNMOUNTED
"device"	B_INT32_TYPE	The **dev_t** number of the unmounted device.

Be careful with the device number: **dev_t**s are quickly recycled. You should only need this number if you're keeping a list of the **dev_t**s of all mounted disks and you want to remove the **dev_t** for this recently-unmounted volume (keeping in mind that a device mounted message bearing this **dev_t** may arrive in the meantime).

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Storage Kit Table of Contents

- The Storage Kit
- File System Architecture
- Entries and Nodes
- MIME and File Types
- BAppFileInfo
- BDirectory
- BEntry
- BEntryList
- BFile
- BFilePanel
- BMimeType
- BNode
- BNodeInfo
- The Node Monitor
- BPath
- BQuery
- BRefFilter
- BResources
- BStatable
- BSymLink
- BVolume
- BVolumeRoster
- Global C Functions
- File System Info C Functions
- Attribute C Functions
- Index C Functions

Query C Functions
Global Constants and Defined Types

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Storage Kit: Master Index

!

!=	BEntry
!=	BNode
!=	BPath
!=	BVolume

<

Navigating the File System	File System Architecture
Symbolic Links	File System Architecture
The General Approach.	BEntry
B_ENTRY_CREATED	BDirectory
B_ENTRY_MOVED	BDirectory
B_ENTRY_REMOVED	BDirectory

=

=	BDirectory
=	BEntry
=	BFile
=	BNode
=	BPath

=	BVolume
==	BEntry
==	BMimeType
==	BNode
==	BPath
==	BVolume

A

Abstract Entries	BEntry
Abstract Entries	BStatable
Access to Directories and Symbolic Links	BFile
Accessing Unreadable and Unwritable Entries	BStatable
AddResource()	BResources
AllowsTypeCode()	BPath
An Example	Attribute C Functions
An Example	Query C Functions
BAppFileInfo	BAppFileInfo
BAppFileInfo()	BAppFileInfo
~BAppFileInfo()	BAppFileInfo
BAppFileInfo	BNodeInfo
Append()	BPath
B_ATTR_CHANGED	The Node Monitor
attr_info	BNode

Attribute C Functions	Attribute C Functions
Attribute C Functions	Attribute C Functions
Attribute Functions	Attribute C Functions
Attributes	BQuery
Attributes, Resources, and the File Type Database	BAppFileInfo

---

B

A BDirectory Knows its Entry	Entries and Nodes
The BPath Calling Convention	BPath

C

C Functions	BDirectory
The C Functions	BFilePanel
C Functions	BMimeType
C Functions	File System Info C Functions
Cancel Notification	BFilePanel
Capacity()	BVolume
Choosing an Iterator	BEntryList
Clear()	BQuery
Combining Flags and the 4096 Limit	The Node Monitor

Confirmable Node Flavors	BFilePanel
Constants	Global Constants and Defined Types
Constructing a Predicate	BQuery
Constructing and Fine-tuning the Panel	BFilePanel
Constructor and Destructor	BAppFileInfo
Constructor and Destructor	BDirectory
Constructor and Destructor	BEntry
Constructor and Destructor	BFilePanel
Constructor and Destructor	BFile
Constructor and Destructor	BMimeType
Constructor and Destructor	BNodeInfo
Constructor and Destructor	BNode
Constructor and Destructor	BPath
Constructor and Destructor	BQuery
Constructor and Destructor	BResources
Constructor and Destructor	BSymLink
Constructor and Destructor	BVolumeRoster
Constructor and Destructor	BVolume
Contains()	BDirectory
Contains()	BMimeType
Converting a BNode to an Instance of a Derived Class	BNode
Converting a BPath	BPath
Converting to BDirectory	BNode

Converting to BFile or BSymLink	BNode
CountEntries()	BDirectory
CountEntries()	BEntryList
CountEntries()	BQuery
CreateDirectory()	BDirectory
CreateFile()	BDirectory
CreateSymLink()	BDirectory
create_app_meta_mime	BMimeType
create_directory()	BDirectory
Creating a File From an Abstract Entry	BEntry
Creating and Using a BFilePanel	BFilePanel
Creating New Directories	BDirectory

D

Data Format	BResources
Data Function	BResources
Data Ownership	BResources
Default Root Nodes	File System Architecture
Defined Types	Global Constants and Defined Types
Delete()	BMimeType
Derived Classes and their Uses	BNode
dev_for_path()	File System Info C Functions
Device()	BVolume
B_DEVICE_MOUNTED	The Node Monitor

B_DEVICE_UNMOUNTED	The Node Monitor
Directories are Persistent, Names Are Not	BEntry
Directories at Root	File System Architecture
Directories	Global Constants and Defined Types
BDirectory	BDirectory
BDirectory()	BDirectory
~BDirectory()	BDirectory
The dirent Structure and GetNextDirents()	BEntryList

E

BEntries and Locked Nodes	BEntry
Entries and Nodes	Entries and Nodes
Entries and Nodes	Entries and Nodes
Entries	Entries and Nodes
BEntry	BEntry
BEntry()	BEntry
~BEntry	BEntry
Entry Created	BQuery
BEntryList	BEntryList
BEntryList	BEntryList
The Entry List Pointer	BEntryList
Entry Removed	BQuery
B_ENTRY_CREATED	The Node Monitor

B_ENTRY_MOVED	The Node Monitor
entry_ref	Global Constants and Defined Types
B_ENTRY_REMOVED	The Node Monitor
Errors	BAppFileInfo
Errors	BNodeInfo
Executables as Resource Files	BResources
Exists()	BEntry

F

Fetch()	BQuery
BFile	BFile
BFile()	BFile
~BFile()	BFile
File()	BResources
File Descriptors	BEntry
File Open Mode Constants	Global Constants and Defined Types
BFilePanel	BFilePanel
BFilePanel()	BFilePanel
~BFilePanel()	BFilePanel
File System Architecture	File System Architecture
File System Architecture	File System Architecture
File System Handlers	File System Architecture
File System Info C Functions	File System Info C Functions

File System Info C Functions	File System Info C Functions
File System Layout and the Root File System	File System Architecture
Filter()	BRefFilter
FindEntry()	BDirectory
find_directory()	BDirectory
Finding a Directory	BDirectory
Finding Views in the Panel	BFilePanel
Flatten()	BPath
BFlattenable Functions	BPath
FlattenedSize()	BPath
Forcing Initialization	BPath
FreeBytes()	BVolume
fs_close_attr_dir	Attribute C Functions
fs_close_index_dir()	Index C Functions
fs_close_query()	Query C Functions
fs_create_index()	Index C Functions
fs_fopen_attr_dir()	Attribute C Functions
fs_open_attr_dir()	Attribute C Functions
fs_open_index_dir()	Index C Functions
fs_open_query()	Query C Functions
fs_read_attr	Attribute C Functions
fs_read_attr_dir	Attribute C Functions
fs_read_index_dir()	Index C Functions
fs_read_query()	Query C Functions
fs_remove_attr()	Attribute C Functions

fs_remove_index()	Index C Functions
fs_rewind_attr_dir()	Attribute C Functions
fs_rewind_index_dir()	Index C Functions
fs_stat_attr	Attribute C Functions
fs_stat_dev()	File System Info C Functions
fs_stat_index()	Index C Functions
fs_write_attr()	Attribute C Functions
Functions Inherited From BNodeInfo	BAppFileInfo

G

Generative Functions	BResources
GetAccessTime()	BStatable
GetAppFlags()	BAppFileInfo
GetAppHint()	BMimeType
GetAppHint()	BNodeInfo
GetAttrInfo()	BMimeType
GetAttrInfo()	BNode
GetBootVolume()	BVolumeRoster
GetCreationTime()	BStatable
GetEntry()	BDirectory
GetFileExtensions()	BMimeType
GetGroup()	BStatable
GetIcon()	BAppFileInfo
GetIcon()	BMimeType

GetIcon()	BNodeInfo
GetIcon()	BVolume
GetIconForType()	BAppFileInfo
GetIconForType()	BMimeType
GetInstalledSupertypes()	BMimeType
GetInstalledTypes()	BMimeType
GetLongDescription()	BMimeType
GetModificationTime()	BStatable
GetName()	BEntry
GetName()	BVolume
GetNextAttrName()	BNode
GetNextDirents()	BDirectory
GetNextDirents()	BEntryList
GetNextDirents()	BQuery
GetNextEntry()	BDirectory
GetNextEntry()	BEntryList
GetNextEntry()	BQuery
GetNextRef()	BDirectory
GetNextRef()	BEntryList
GetNextRef()	BQuery
GetNextSelectedRef()	BFilePanel
GetNextVolume()	BVolumeRoster
GetNodeRef()	BStatable
GetOwner()	BStatable
GetPanelDirectory()	BFilePanel
GetParent()	BEntry

GetParent()	BPath
GetPath()	BEntry
GetPermissions()	BStatable
GetPredicate()	BQuery
GetPreferredApp()	BAppFileInfo
GetPreferredApp()	BMimeType
GetPreferredApp()	BNodeInfo
GetRef()	BEntry
GetResourceInfo()	BResources
GetRootDirectory()	BVolume
GetShortDescription()	BMimeType
GetSignature()	BAppFileInfo
GetSize()	BFile
GetSize()	BStatable
GetStat()	BEntry
GetStat()	BStatable
GetStatFor()	BDirectory
GetSupertype()	BMimeType
GetSupportedTypes()	BAppFileInfo
GetSupportingApps()	BMimeType
GetTrackerIcon()	BNodeInfo
GetType()	BAppFileInfo
GetType()	BNodeInfo
GetVersionInfo()	BAppFileInfo
GetWildcardApps()	BMimeType
get_device_icon	BMimeType

get_ref_for_path()	BEntry
Getting a dirent	BEntryList
Global C Function	BEntry
Global C Functions	Global C Functions
Global C Functions	Global C Functions
Global Constants and Defined Types	Global Constants and Defined Types
Global Constants and Defined Types	Global Constants and Defined Types

---

H

HasResource()	BResources
Hide()	BFilePanel
Hide When Done	BFilePanel
HidesWhenDone()	BFilePanel
Hierarchical Organization	File System Architecture
Hook Functions	BFilePanel
Hook Functions	BRefFilter

I

Icon Sizes	Global Constants and Defined Types
Identifying a Resource within a Resource File	BResources
Identifying and Creating Resource	BResources

Files	
Immutability	BPath
Implications	BNode
Index C Functions	Index C Functions
Index C Functions	Index C Functions
The Index Directory	Index C Functions
Index Functions	Index C Functions
Info Functions	BResources
InitCheck()	BAppFileInfo
InitCheck()	BEntry
InitCheck()	BFile
InitCheck()	BMimeType
InitCheck()	BNodeInfo
InitCheck()	BNode
InitCheck()	BPath
InitCheck()	BVolume
Initialization and File Descriptors	BSymLink
Initialization	BAppFileInfo
Initialization	BNodeInfo
Initializing a BResources Object	BResources
Initializing a BVolume	BVolume
Initializing and Normalizing	BPath
Initializing and Opening	BFile
Initializing and Traversing	BEntry
Install()	BMimeType
Installing and Removing Indices	Index C Functions

IsAbsolute()	BSymLink
IsDirectory()	BStatable
IsFile()	BStatable
IsFixedSize()	BPath
IsInstalled()	BMimeType
IsLive()	BQuery
Is Modal?	BFilePanel
IsPersistent()	BVolume
IsReadOnly()	BVolume
IsReadable()	BFile
IsRemovable()	BVolume
IsRootDirectory()	BDirectory
IsShared()	BVolume
IsShowing()	BFilePanel
IsSupertypeOnly()	BMimeType
IsSupportedType()	BAppFileInfo
IsSymLink()	BStatable
IsUsingAttributes()	BAppFileInfo
IsUsingResources()	BAppFileInfo
IsValid()	BMimeType
IsWritable()	BFile

K

KnowsAttr()	BVolume
KnowsMime()	BVolume

KnowsQuery()	BVolume

L

Leaf()	BPath
Lies	Entries and Nodes
Limits Constants	Global Constants and Defined Types
Live Queries	BQuery
LoadResource()	BResources
Lock()	BNode

M

MIME and File Types	MIME and File Types
MIME and File Types	MIME and File Types
MIME in the BeOS	MIME and File Types
MakeLinkedPath()	BSymLink
Member Functions	BAppFileInfo
Member Functions	BDirectory
Member Functions	BEntryList
Member Functions	BEntry
Member Functions	BFilePanel
Member Functions	BFile
Member Functions	BMimeType
Member Functions	BNodeInfo
Member Functions	BNode

Member Functions	BPath
Member Functions	BQuery
Member Functions	BRefFilter
Member Functions	BResources
Member Functions	BStatable
Member Functions	BSymLink
Member Functions	BVolumeRoster
Member Functions	BVolume
MergeFrom()	BResources
Messenger()	BFilePanel
Messenger()	BVolumeRoster
BMimeType	BMimeType
BMimeType()	BMimeType
~BMimeType()	BMimeType
Modifying the Look of the File Panel	BFilePanel
Monitor Flags	The Node Monitor
More Facts	Entries and Nodes
Mount and Unmount	BVolume
MoveTo()	BEntry
Multiple Retrieval	BEntryList
Multiple Selection	BFilePanel

N

next_dev()	File System Info C Functions
BNode	BNode

BNode()	BNode
~BNode()	BNode
A Node can Lose its Entry	Entries and Nodes
Node Flavors	Global Constants and Defined Types
BNodeInfo	BNodeInfo
BNodeInfo()	BNodeInfo
~BNodeInfo()	BNodeInfo
Node Info equals Attributes	BNodeInfo
BNode Instances	BNode
Node Locking	BNode
Node Monitor Functions	The Node Monitor
Node Monitor Opcodes	Global Constants and Defined Types
The Node Monitor	The Node Monitor
The Node Monitor	The Node Monitor
Node Monitoring a Directory	BDirectory
The "Node Pool" is Limited (File Descriptors)	BNode
node_ref	Global Constants and Defined Types
Nodes and Entries	BStatable
Nodes are Dumb	BNode
Nodes	Entries and Nodes
Normalization by Default?	BPath
Notification Message	BFilePanel
Notification Messages	The Node Monitor

O

Opcode Constants	The Node Monitor
Open Notification	BFilePanel
Opening, Reading, and Closing a Query	Query C Functions
Operators	BDirectory
Operators	BEntry
Operators	BFile
Operators	BMimeType
Operators	BNode
Operators	BPath
Operators	BVolume
Other Details	BStatable
Other Normalization Details	BPath
Overview	BRefFilter

P

Panel Directory	BFilePanel
Panel Mode	BFilePanel
PanelMode()	BFilePanel
Parsing and Tricks	The Node Monitor
Passing a BPath in a BMessage	BPath
BPath	BPath
Path()	BPath

~BPath	BPath
Position()	BFile
PredicateLength()	BQuery
The Predicate, Attributes, and Indices	BQuery
PreloadResourceType()	BResources
PushAttr()	BQuery
PushDouble()	BQuery
PushFloat()	BQuery
PushInt32()	BQuery
PushInt64()	BQuery
PushOp()	BQuery
PushString()	BQuery
PushUInt32()	BQuery
PushUInt64()	BQuery

Q

BQuery	BQuery
BQuery()	BQuery
~BQuery()	BQuery
Query C Functions	Query C Functions
Query C Functions	Query C Functions
Query Functions	Query C Functions
Query Operation Constants	Global Constants and Defined Types
Query Update Messages	BQuery

query_op	BQuery

R

ReadAt()	BFile
ReadAttr()	BNode
Read()	BFile
ReadLink()	BSymLink
Reading and Writing a Resource File as a Plain File	BResources
Ref Filter	BFilePanel
RefFilter()	BFilePanel
BRefFilter	BRefFilter
BRefFilter	BRefFilter
Refresh()	BFilePanel
Relationship to stat()	BStatable
RemoveAttr()	BNode
Remove()	BEntry
RemoveResource()	BResources
RenameAttr()	BNode
Rename()	BEntry
Resource Data	BResources
BResources	BResources
BResources()	BResources
~BResources()	BResources
Retrieving Entries	BDirectory

Reusing your BQuery	BQuery
RewindAttrs()	BNode
Rewind()	BDirectory
Rewind()	BEntryList
Rewind()	BFilePanel
Rewind()	BQuery
Rewind()	BVolumeRoster

S

Save Notification	BFilePanel
Seek()	BFile
SelectionChanged()	BFilePanel
SendMessage()	BFilePanel
SetAccessTime()	BStatable
SetAppFlags()	BAppFileInfo
SetAppHint()	BMimeType
SetAppHint()	BNodeInfo
SetAttrInfo()	BMimeType
SetButtonLabel()	BFilePanel
SetCreationTime()	BStatable
SetFileExtensions()	BMimeType
SetGroup()	BStatable
SetHideWhenDone()	BFilePanel
SetIcon()	BAppFileInfo
SetIcon()	BMimeType

SetIcon()	BNodeInfo
SetIconForType()	BAppFileInfo
SetIconForType()	BMimeType
SetInfoLocation()	BAppFileInfo
SetLongDescription()	BMimeType
SetMessage()	BFilePanel
SetModificationTime()	BStatable
SetOwner()	BStatable
SetPanelDirectory()	BFilePanel
SetPermissions()	BStatable
SetPredicate()	BQuery
SetPreferredApp()	BAppFileInfo
SetPreferredApp()	BMimeType
SetPreferredApp()	BNodeInfo
SetRefFilter()	BFilePanel
SetSaveText()	BFilePanel
SetShortDescription()	BMimeType
SetSignature()	BAppFileInfo
SetSize()	BFile
SetSupportedTypes()	BAppFileInfo
SetTarget()	BFilePanel
SetTarget()	BQuery
SetTo()	BAppFileInfo
SetTo()	BDirectory
SetTo()	BEntry
SetTo()	BFile

SetTo()	BMimeType
SetTo()	BNodeInfo
SetTo()	BNode
SetTo()	BPath
SetTo()	BResources
SetTo()	BVolume
SetType()	BAppFileInfo
SetType()	BNodeInfo
SetVersionInfo()	BAppFileInfo
SetVolume()	BQuery
Show()	BFilePanel
StartWatching()	BMimeType
StartWatching()	BVolumeRoster
The stat Structure	BStatable
B_STAT_CHANGED	The Node Monitor
BStatable	BStatable
BStatable	BStatable
StopWatching()	BMimeType
StopWatching()	BVolumeRoster
stop_watching()	The Node Monitor
The Storage Kit	The Storage Kit
The Storage Kit	The Storage Kit
Subtleties and Details	BEntry
Supports()	BAppFileInfo
BSymLink	BSymLink
BSymLink()	BSymLink

~BSymLink()	BSymLink
Symlinks at Root	File System Architecture
Sync()	BNode
Sync()	BResources

T

Talents and Abilities	BEntry
The Target and the Messages it Sees	BFilePanel
Target	BFilePanel
Traverso Post Facto	BEntry
true	BEntryList
Type()	BMimeType
TypeCode()	BPath

U

Unflatten()	BPath
Unlock()	BNode
Unset()	BDirectory
Unset()	BEntry
Unset()	BFile
Unset()	BMimeType
Unset()	BNode
Unset()	BPath
Unset()	BVolume

update_mime_info	BMimeType

V

Valid MIME Strings	BMimeType
Values	BQuery
Version Kinds	Global Constants and Defined Types
version_info	Global Constants and Defined Types
Virtual File Systems	File System Architecture
BVolume	BVolume
BVolume()	BVolume
~BVolume	BVolume
BVolumeRoster	BVolumeRoster
BVolumeRoster()	BVolumeRoster
~BVolumeRoster()	BVolumeRoster

W

WasHidden()	BFilePanel
watch_node() Flags	Global Constants and Defined Types
watch_node()	The Node Monitor
What's Really Going On	BPath
When to Traverse	BEntry
Window()	BFilePanel
WriteAt()	BFile
WriteAttr()	BNode

Write()	BFile
WriteTo()	BResources

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BEntry

Derived from: BStatable
Declared in: be/storage/Entry.h
Library: libbe.so
Summary: *more...*

The BEntry class defines objects that represent "locations" in the file system hierarchy. Each location (or *entry*) is given as a name within a directory. For example, when you create a BEntry thus...

```
BEntry entry("/boot/home/fido");
```

...you're telling the BEntry object to represent the *location* of the file called **fido** within the directory "/boot/home".

A BEntry doesn't care whether the entry you tell it to represent is a plain file, a directory, or a symbolic link—it doesn't even care if the entry even *exists* (but we'll get to that later in "Abstract Entries"):

- All the BEntry cares about is a *name* in a *directory*.

The most important implication of this is the object's attitude towards data. **BEntries don't know how to operate on data**. You can't use a BEntry to read or write a file's data or attributes. For data operations, you have to turn your BEntry into a BNode.

Nonetheless, it's often convenient to speak of a BEntry as *having* data; for example, the phrase "the entry's data" really means "the data that lies in the file that's located by the entry."

Talents and Abilities

A properly initialized BEntry object (we'll get to the rules of initialization later) knows the following:

- **Location info.** A BEntry knows its own (leaf) name (`GetName()`), its full pathname (`GetPath()`), and the identity of its parent directory (`GetParent()`).
- **BStatable info.** As a descendant of BStatable, a BEntry can return statistical information about the entry's data—its size, creation date, owner, and so on.
- `entry_ref` **identifier.** A BEntry can return the `entry_ref` that globally identifies the entry (`GetRef()`).

A BEntry can do these things:

- **Perform hierarchical operations**. A BEntry can change the name of its entry (`Rename()`), move it to another directory (`Move()`), and remove it from the file hierarchy (`Remove()`).

- **Initialize BNode objects.** The constructors and `SetTo()` initializers for BNode and its children (BFile, BDirectory, and BSymLink) accept BEntry arguments.

As mentioned above, the most important thing that a BEntry *can't* do is access its own data: **A BEntry can't read or write data or attributes.** To do these things you need a BNode object.

(Actually, this isn't entirely true: A BEntry *can* set the size of its data through the **`BStatable::SetSize()`** function. The function only works on plain files.)

Initializing and Traversing

To initialize a BEntry, you have to tell it which entry to represent; in other words, you have to identify a directory and a name. You can initialize a BEntry object directly...

- during construction,
- through the **`SetTo()`** function,
- or through the assignment operator.

Or you can have some other object initialize your BEntry for you, by passing the BEntry as an argument to...

- BDirectory's **`FindEntry()`** or **`GetEntry()`** function,
- BEntryList's **`GetNextEntry()`** function (implemented by BDirectory and BQuery).
- BEntry's **`GetParent()`** function.

In all cases (except the assignment operator) you're asked if you want to "traverse" the entry during initialization. Traversal is used to "resolve" symbolic links:

- **If you traverse**: The BEntry will point to the entry that the symbolic link is linked to.
- **If you don't traverse**: The BEntry will point to the symbolic link itself.

For example, let's say **/boot/home/fidoLink** is linked to **/fido**, to wit:

```
$ cd /boot/home
$ ln -s ./fido fidoLink
```

Now let's make a traversed BEntry for **fidoLink**:

```
/* The second argument is the traversal bool. */
BEntry entry("/boot/home/fidoLink", true);
```

If we ask for the entry's pathname...

```
BPath path;
entry.GetPath(&path);
printf("Pathname: %sn", path.Path());
```

...we see

```
Pathname: /boot/home/fido
```

In other words, the BEntry refers to **fido**, not **fidoLink.**

Traversal resolves nested links—it really wants to find a "real" file (or directory). If the entry that you're initializing to isn't a link, then the traversal flag is ignored.

When to Traverse

When should you traverse, and when not? Here are a few rules of thumbs:

- If somebody hands you a file reference—if your app gets a **`RefsReceived()`** message—then you probably want to traverse the entry.
- If you're pawing over the contents of a directory (through BDirectory's **`GetNextEntry()`**), then you probably don't want to traverse.
- If you're looking at the result of a query (through BQuery's **`GetNextEntry()`**), then you almost *certainly* don't want to traverse. The query finds entries that satisfy certain criteria; if a symbolic link is in the list, it's because the link itself was a winner. If the linked-to file is also a winner, it will show up on its own.

Traverso Post Facto

Let's say you create a BEntry (to a symlink) without traversing, but then you decide that you *do* want to resolve the link. Unfortunately, you can't resolve in-place; instead, you have to initialize another BEntry using info (**`entry_ref`** or pathname) that you get from the link entry:

```
BEntry entry1("/boot/home/fidoLink", false);
BEntry entry2;
entry_ref ref;

/* First we check to see if it's a link. */
if (entry1.IsSymLink()) {
   /* Get the link's entry_ref... */
   entry1.GetRef(&ref);

   /* ...and use it to initialize the other BEntry. */
   entry2.SetTo(&ref, true);
}
```

Abstract Entries

As we all should know by now, a BEntry identifies a name within a specific directory. The directory that a BEntry identifies must exist, but the entry that corresponds to the name doesn't have to. In other words...

- A BEntry can represent a file that doesn't exist. The entry is said to be "abstract."

For example, the following construction creates a BEntry object based on a BDirectory and a name:

```
BEntry entry(someDir, "myFile.h");
```

Let's assume that **myFile.h** doesn't exist. As long as the directory that's referred to by `someDir` *does* exist, then the construction is legal. Some of the BEntry functions (those inherited from BStatable, for instance) won't work, but the object itself is valid.

But validity doesn't equal existence:

- `SetTo()` and `InitCheck()` ***do** not* tell you if a BEntry's entry actually exists. Don't be confused; a return value of `B_OK` simply means the object is valid.

If you want to know if a BEntry's entry actually exists, use the `Exists()` function.

Creating a File From an Abstract Entry

To turn an abstract BEntry into a real entry (or, more accurately, a real node), you have to specify the flavor of node that you want. There are two methods for creating a node; the first is general, the second applies to plain files only.

The General Approach.

BDirectory's `CreateFile()`, `CreateDirectory()`, and `CreateSymLink()` functions create nodes of the designated flavor. The functions don't take BEntry arguments directly; instead, you invoke the functions on the BEntry's directory, passing the entry's leaf name as an argument. Here we turn an abstract entry (`entry`) into a directory:

```
BPath path;
char name[B_FILE_NAME_LENGTH]; /* A buffer for the name. */
BDirectory parent; /* The parent of our entry. */
BDirectory target_dir; /* The product of the transformation. */

if (!entry.Exists()) {
   entry.GetParent(&path);
   entry.GetName(name);
   parent.SetTo(&path);
   parent.CreateDirectory(name, &dir);
}
```

The Plain-File-Only Approach.

You can create a plain file by passing the BEntry to the BFile constructor or `SetTo()` function. To do this, you also have to add `B_CREATE_FILE` to the "open mode" flags:

```
BFile file;
```

```
if (!entry.Exists())
   file.SetTo(&entry, B_CREATE_FILE|B_READ_WRITE);
```

Subtleties and Details

The following details understand you should, particularly if you want to participate in bedevtalk.

File Descriptors

Although it's not intuitively obvious, a BEntry object *does* consume a file descriptor. The file descriptor is opened on the entry's directory.

Your app has a limited number of file descriptors (currently 128, max), so you may not want to cache BEntry objects as your primary means for identifying an entry. If you're going to be dealing with a lot of entries and you want to keep track of them all, it's better to cache `entry_ref` structures or BPath objects.

Directories are Persistent, Names Are Not

One more time: A BEntry identifies an entry as a name in a directory. As described above, the directory is maintained internally as a file descriptor; the name is simply a string. This means that...

- **The directory for a given BEntry is persistent. If you move the directory, the file descriptor, and so the BEntry, moves with it.**
- **The name isn't persistent. If the user renames the leaf that a BEntry is pointing to, the BEntry will become abstract.**

For example, take the following BEntry...

```
BEntry entry("/boot/home/lbj/footFetish.jpeg");
```

If the user moves the directory...

```
$ cd /boot/home
$ mv lbj jfk
```

The BEntry (`entry`) "moves" with the directory. If you print the pathname and ask if the BEntry's entry exists...

```
BPath path;
entry.GetPath(&path);
printf("> Foot movie: %sn", path.Path());
printf("> Exists? %sn", entry.Exists()?"Oui":"Non");
```

...you'll see this:

```
> Foot movie: /boot/home/jfk/footFetish.jpeg
> Exists? Oui
```

The same isn't so for the name portion of a BEntry. If the user now moves footFetish.jpeg...

```
$ cd /boot/home/jfk
$ mv footFetish.jpeg hammerToe.jpeg
```

...your BEntry *will not* follow the file (it doesn't "follow the data"). The object will still represent the entry called footFetish.jpeg. The BEntry will, in this case, become abstract.

Don't be confused: The BEntry only "loses track" of a renamed entry if the name change is made behind the object's back. Manipulating the entry name through the BEntry object's `Rename()` function (for example), doesn't baffle the object. For example:

```
BPath path;
BEntry entry("/boot/home/lbj/footFetish.jpeg");

entry.Rename("hammerToe.jpeg");
entry.GetPath(&path);
printf("> Foot movie: %sn", path.Path());
printf("> Exists? %sn", entry.Exists()?"Oui":"Non");
```

...and we see...

```
> Foot movie: /boot/home/lbj/hammerToe.jpeg
> Exists? Oui
```

BEntries and Locked Nodes

You can't lock an entry, but you can lock the entry's node (through BNode's `Lock()` function). Initializing a BEntry to point to a locked node is permitted, but the entry's directory must *not be locked.* If the directory is locked, the BEntry constructor and `SetTo()` function fail and set `InitCheck()` to `B_BUSY`.

Furthermore, the destination directories in BEntry's `Rename()` and `MoveTo()` must be unlocked for the functions to succeed. And *all* directories in the path to the entry must be unlocked for `GetPath()` to succeed.

If you get a `B_BUSY` error, you may want to try again—it's strongly advised that locks be held as briefly as possible.

Constructor and Destructor

BEntry()

```
BEntry(const BDirectory *dir, const char *path, bool traverse = false)
BEntry(const entry_ref *ref, bool traverse = false)
BEntry(const char *path, bool traverse = false)
BEntry(void)
BEntry(const BEntry &entry)
```

Creates a new BEntry object that represents the entry described by the arguments. See the analogous `SetTo()` functions for descriptions of the flavorful constructors.

The default constructor does nothing; it should be followed by a call to `SetTo()`.

The copy constructor points the new object to the entry that's represented by the argument. The two objects themselves maintain separate representation of the entry; in other words, they each contain their own a) file descriptor and b) string to identify the entry's a) directory and b) name.

To see if the initialization was successful, call `InitCheck()`.

~BEntry

```
virtual ~BEntry()
```

Closes the BEntry's file descriptor and destroys the BEntry object.

Member Functions

Exists()

```
bool Exists(void) const
```

Returns `true` if the entry exists; `false` otherwise.

GetName() , GetPath()

```
status_t GetName(char *buffer) const
status_t GetPath(BPath *path) const
```

These functions return the leaf name and full pathname of the BEntry's entry. The arguments must be allocated before they're passed in.

`GetName()` copies the leaf name into *buffer.* The buffer must be large enough to accommodate the name; `B_FILE_NAME_LENGTH` is a 100% safe bet:

```
char name[B_FILE_NAME_LENGTH];
entry.GetName(name);
```

If `GetName()` fails, **buffer* is pointed at `NULL`.

`GetPath()` takes the entry's full pathname and initializes the BPath argument with it. To retrieve the path from the BPath object, call `BPath::Path()`:

```
BPath path;
entry.GetPath(&path);
printf(">Entry pathname: %sn", path.Path());
```

If `GetPath()` fails, the argument is `Unset()`.

RETURN CODES

- `B_OK`. The information was successfully retrieved.
- `B_NO_INIT`. The BEntry isn't initialized.
- `B_BUSY` (`GetPath()` only). A directory in the entry's path is locked.

GetParent()

```
status_t GetParent(BEntry *entry) const
```

status_t GetParent(BDirectory *dir) const

Gets the directory, as a BEntry or BDirectory object, in which the object's entry lives. The argument must be allocated before it's passed in.

If the function is unsuccessful, the argument is Unset(). Because of this, you should be particularly careful if you're using the BEntry-argument version to destructively get a BEntry's parent:

```
if (entry.GetParent(&entry) != B_OK) {
   /* you just lost 'entry' */
}
```

This example *is* legal; for example, you can use destructive iteration to loop your way up to the root directory. When you reach the root ("/"), GetParent() returns B_ENTRY_NOT_FOUND:

```
BEntry entry("/boot/home/fido");
status_t err;
char name[B_FILE_NAME_LENGTH];

/* Spit out the path components backwards, one at a time. */
do {
   entry.GetName(name);
   printf("> %sn", name);
} while ((err=entry.GetParent(&entry)) == B_OK);

/* Complain for reasons other than reaching the top. */
if (err != B_ENTRY_NOT_FOUND)
   printf(">> Error: %sn", strerror(err));
```

This produces:

```
> fido
> home
> boot
> /
```

RETURN CODES

- **B_OK. The information was successfully retrieved.**
- **B_NO_INIT. This BEntry isn't initialized.**
- **B_ENTRY_NOT_FOUND. Attempt to get the parent of the root directory.**
- **B_NO_MORE_FDS. Couldn't get another file descriptor.**

GetPath() see GetName()

GetRef()

```
status_t GetRef(entry_ref *ref) const
```

Gets the entry_ref for the object's entry; *ref* must be allocated before it's passed in. As with BEntry objects, entry_ref structures can be abstract—getting a valid entry_ref does *not* guarantee that the entry actually exists.

If the function isn't successful, *ref* is unset.

RETURN CODES

- B_OK. The entry_ref was successfully retrieved.
- B_NO_INIT. This object isn't initialized.
- B_NO_MEMORY. Storage for the entry_ref's name couldn't be allocated.

GetStat()

```
virtual status_t GetStat(struct stat *st) const
```

GetStat() returns the stat structure for the entry. The structure is copied into the *st* argument, which must be allocated. The BStatable object does *not* cache the stat structure; every time you call GetStat(), fresh stat information is retrieved.

RETURN CODES

- B_OK. Success.
- B_NO_MEMORY. Couldn't get the necessary resources to complete the transaction.
- B_BAD_VALUE. The entry doesn't exist (abstract entry).

InitCheck()

```
status_t InitCheck(void) const
```

Returns the status of the previous construction, assignment operation, or SetTo() call.

RETURN CODES

- B_OK. The initialization was successful.
- B_NO_INIT. The object is uninitialized (this includes Unset()).
- See SetTo() for other errors.

MoveTo() see Rename()

Remove()

status_t Remove(void)

Remove() "unlinks" the entry from its directory. The entry's node isn't destroyed until all file descriptors that are open on the node are closed. This means that if you create BFile based on a BEntry, and then Remove() the BEntry, the BFile will still be able to read and write the file's data—the BFile has no way of knowing that the entry is gone. When the BFile is deleted, the node will be destroyed as well.

Remove() does *not* invalidate the BEntry. It simply makes it abstract (see "Abstract Entries").

RETURN CODES

- B_OK. Success.
- B_NO_INIT. The BEntry is not initialized.
- B_BUSY. The entry's directory is locked.

Rename() , MoveTo()

status_t Rename(const char *path, bool clobber = false)
status_t MoveTo(BDirectory *dir, const char *path = NULL, bool clobber = false)

These functions move the BEntry's entry and node to a new location. In both cases, the BEntry must *not* be abstract—you can't rename or move an abstract entry.

Rename() moves the entry to a new name, as given by *path*. *path* is usually a simple leaf name, but it can be a relative path. In the former case (simple leaf) the entry is renamed within its current directory. In the latter, the entry is moved into a subdirectory of its current directory, as given by the argument.

MoveTo() moves the entry to a different directory and optionally renames the leaf. Again, *path* can be a simple leaf or a relative path; in both cases, *path* is reckoned off of *dir*. If path is NULL, the entry is moved to *dir*, but retains its old leaf name.

If the entry's new location is already taken, the *clobber* argument decides whether the existing entry is removed to make way for yours. If it's true, the existing entry is removed; if it's false, the Rename() or MoveTo() function fails.

Upon success, this is updated to reflect the change to its entry. For example, when you invoke Rename() on a BEntry, the name of *that specific* BEntry object also changes. If the

rename or move-to isn't successful, `this` isn't altered.

RETURN CODES

- **`B_OK`. Success.**
- **`B_NO_INIT`. The BEntry is not initialized.**
- **`B_ENTRY_NOT_FOUND`. A directory to the new location doesn't exist, or this is an abstract entry.**
- **`B_FILE_EXISTS`. The new location is already taken (and you're not clobbering).**
- **`B_BUSY`. The directory that you're moving the entry into is locked.**

SetTo() , Unset()

status_t `SetTo`(const entry_ref *_ref_, bool _traverse_ = false) **status_t `SetTo`(const const char *_path_, bool _traverse_ = false)** **status_t `SetTo`(const BDirectory *_dir_, const char *_path_, bool _traverse_ = false)**
void `Unset`(void)

Frees the BEntry's current entry reference, and initializes it to refer to the entry identified by the argument(s):

- **In the *ref* version, the BEntry is initialized to refer to the given `entry_ref`.**
- **In the *path* version, *path* can be absolute or relative, and can contain "." and ".." elements. If *path* is relative, it's reckoned off of the current working directory.**
- **In the *dir*/*path* version, *path* must be relative. It's reckoned off of the directory given by *dir*.**

The *traverse* argument is used to resolve (or not) entries that are symlinks:

- **If *traverse* is `true`, the link is resolved.**
- **If *traverse* is `false`, the BEntry refers to the link itself.**

See " "Initializing and Traversing" on page40" for more information.

When you initialize a BEntry, you're describing a leaf name within a directory. The directory *must* exist, but the leaf doesn't have to. This allows you to create a BEntry to a file that doesn't exist (yet). See " "Abstract Entries" on page42" for more information.

Remember—successfully initializing a BEntry consumes a file descriptor. When you re-initialize, the old file descriptor is closed.

`Unset()` removes the object's association with its current entry, and sets `InitCheck()` to `B_NO_INIT`.

RETURN CODES

- `B_OK`. The BEntry was successfully initialized.
- `B_BAD_VALUE`. Bad argument value; uninitialized *ref* or *dir*.
- `B_ENTRY_NOT_FOUND`. A directory in the path to the entry doesn't exist.
- `B_BUSY`. The entry's directory is locked.

Unset() see SetTo()

Operators

= (assignment)

```
BEntry& operator=(const BEntry &entry)
```

In the expression

```
BEntry a = b;
```

BEntry *a* is initialized to refer to the same entry as *b*. To gauge the success of the assignment, you should call `InitCheck()` immediately afterwards. Assigning a BEntry to itself is safe.

Assigning from an uninitialized BEntry is "successful": The assigned-to BEntry will also be uninitialized (`B_NO_INIT`).

== , != (comparison)

```
bool operator==(const BEntry &entry) const
bool operator!=(const BEntry &entry) const
```

Two BEntry objects are said to be equal if they refer to the same entry (even if the entry is abstract), or if they're both uninitialized.

Global C Function

get_ref_for_path()

status_t get_ref_for_path(const char *_path_, entry_ref *_ref_)

Returns in *ref* an entry_ref for the file specified by the *path* argument.

RETURN CODES

- **B_OK. The *ref* was returned successfully.**
- **B_ENTRY_NOT_FOUND. The file wasn't found, or the pathname string is empty.**
- **B_NO_MEMORY. Not enough memory.**
- **Other file errors.**

The Be Book The Storage Kit The Storage Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStatable

Derived from: *none*
Declared in: be/storage/Statable.h
Library: libbe.so
Summary: *more...*

BStatable is a pure abstract class that provides functionality for its two derived class, BEntry and BNode. The BStatable functions let you get and set "statistical" information about a node in the file system. You can...

- Determine whether the node is a file, directory, or symbolic link.
- Get and set an node's owner, group, and permissions.
- Get and set the node's creation, modification, and access times.
- Get the size of the node's data (*not* counting attributes).
- Get a BVolume object for the node's volume.
- Get the `node_ref` of the node (and pass it to `watch_node()`, most likely).

Nodes and Entries

Technically, BStatable information pertains to *nodes*, not *entries*. The fact that BEntry implements the BStatable functions is a (slightly confusing) convenience: When you invoke a BStatable function on a BEntry object, what you're really doing is asking for information about the node that corresponds to the object.

Abstract Entries

As explained in BEntry, it's possible to create "abstract" BEntry objects; in other words, objects that don't correspond to actual files (nodes) on the disk. You can't get (or set) BStatable information for abstract entries. The BStatable functions return `B_BAD_VALUE` if the invoked-upon entry is abstract.

Relationship to stat()

The BStatable functions are covers for the POSIX `stat()` call. `stat()` retrieves a file-specific `stat` structure, which records the statistics listed above (and then some). Although BStatable was designed to hide `stat` details, you *can* get the `stat` structure through the `GetStat()` function. The `stat` structure is described in "The stat Structure" at the end of this specification.

`stat()` is notorious for being expensive. Furthermore, the `stat` structure is stale as soon as it gets back from the `stat()` call. If you're concerned with efficiency, be aware that *every* BStatable function (the "setters" as well as the "getters") performs a `stat()`. For example,

calling **GetOwner()** and then **GetGroup()** results in two **stat()** calls. If you want to look at lot of fields (within the same **stat** structure) all at once, you might consider using BStatable's **GetStat()** function.

As for integrity, BStatable info-getting functions are obviously in the same boat as the **stat()** call itself: The retrieved data isn't guaranteed to be in sync with the actual state of the **stat()**'d item.

The BDirectory class also defines a **stat**-retrieving function that, in some cases, can be more efficient than the **GetStat()** function defined here:

- The **BDirectory::GetStatFor()** function retrieves the **stat** structure for the node of a named entry within a directory. If you're interested in getting **stat** information for a series of nodes within the same directory, you should use this function. You have to call it iteratively (once for each named entry), but the accumulation of the iterated calls will be faster than the **GetStat()** calls made on the analogous BEntry objects.

Accessing Unreadable and Unwritable Entries

BStatable isn't thwarted by file permissions: If you can construct a valid BEntry or BNode to an item, then you can invoke any of the info-getting BStatable functions on that object:

- The BStatable functions aren't denied even if the node that you're looking at is read-protected. However, you can only invoke the info-setting functions if the node allows writing.
- Similarly, you can get **stat** info for a locked node, but you won't be able to write the info (through functions such as **SetOwner()**) unless your object holds the lock. See BNode for more on locking.

Other Details

You rarely set `stat information`. In practice, you rarely use BStatable's info-setting functions. Setting information such as when a file was created, who owns it, or how big it is, is the responsibility of the system and the privilege of the user. For example, when you **Write()** to a BFile object, the system automatically updates the size and modification date for the file.

Member Functions

GetAccessTime() see GetCreationTime()

GetCreationTime() , SetCreationTime() , GetModificationTime() , SetModificationTime() , GetAccessTime() , SetAccessTime()

status_t **GetCreationTime(**time_t **ctime**) const
status_t **SetCreationTime(**time_t *ctime*)
status_t **GetModificationTime(**time_t **mtime**) const
status_t **SetModificationTime(**time_t *mtime*)
status_t **GetAccessTime(**time_t **atime**) const
status_t **SetAccessTime(**time_t *atime*)

Access time is currently unused.

These function let you get and set the time at which the item was created, last modified, and last accessed (opened). The measure of time is given as seconds since (the beginning of) January 1, 1970.

The time quanta that **stat** uses is *seconds*; the rest of the BeOS measures time in *microseconds* (**bigtime_t**).

RETURN CODES

- **B_OK**. Success.
- **B_NOT_ALLOWED**. You tried to set a time field for a file on a read-only volume.
- **B_NO_MEMORY**. Couldn't get the necessary resources to complete the transaction.
- **B_BAD_VALUE**. The node doesn't exist (abstract entry).

GetGroup() see GetOwner()

GetNodeRef()

status_t **GetNodeRef(**node_ref **nref**) const

Copies the item's **node_ref** structure into the *nref* argument, which must be allocated.

Typically, you use an node's **node_ref** as a key to the Node Monitor by passing the **node_ref** structure to the **watch_node()** function. The Node Monitor watches the node for specific changes; see "The Node Monitor" section of this chapter for details.

As a convenience, you can use a **node_ref** structure to initialize a BDirectory object (through the constructor or **BDirectory::SetTo()** function).

RETURN CODES

- **B_OK**. Success.
- **B_NO_MEMORY**. Couldn't get the necessary resources to complete the transaction.
- **B_BAD_VALUE**. The node doesn't exist (abstract entry).

GetModificationTime() see GetCreationTime()

GetOwner() , SetOwner() , GetGroup() , SetGroup() , GetPermissions() , SetPermissions()

status_t **GetOwner**(uid_t **owner*) const
status_t **SetOwner**(uid_t *owner*)
status_t **GetGroup**(gid_t **group*) const
status_t **SetGroup**(gid_t *group*)
status_t **GetPermissions**(mode_t **perms*) const
status_t **SetPermissions**(mode_t *perms*)

These functions set and get the owner, group, and read/write/execute permissions for the node:

- The *owner* identifier encodes the identity of the user that "owns" the file.
- The *group* identifier encodes the "group" that is permitted group access to the file (as declared by the permissions).
- The *permissions* value records nine "permission facts": Whether the file can be read, written, and executed by the node's owner, by users in the node's group, and by everybody else (read/write/execute * owner/group/others = 9 items).

The **uid_t**, **gid_t**, and **mode_t** types used here are standard POSIX types. All three are 32-bit unsigned integers and are defined in **posix/sys/types.h**.

The *owner* and *group* encodings must match values found in the system's user and group files (which are as currently unimplemented).

The *permissions* value is a combination of the following bitfield constants (defined in **posix/sys/stat.h**):

- **S_IRUSR** owner's read bit.
- **S_IWUSR** owner's write bit.
- **S_IXUSR** owner's execute bit.
- **S_IRGRP** group's read bit.
- **S_IWGRP** group's write bit.
- **S_IXGRP** group's execute bit.
- **S_IROTH** others' read bit.
- **S_IWOTH** others' write bit.
- **S_IXOTH** others' execute bit.

For example:

```
/* Is a file readable by everybody? */
mode_t perms;
if (node.GetPermissions(&perms) < B_OK)
   /* handle the error... */

if (perms & S_ISROTH)
   // Yes it is
else
   // No it isn't
```

RETURN CODES

- **B_OK**. Success.
- **B_NOT_ALLOWED**. You tried to set permissions on a read-only volume.
- **B_BAD_VALUE**. The node doesn't exist (abstract entry).

GetPermissions() see GetOwner()

GetSize()

```
status_t GetSize(off_t *size) const
```

Gets the size of the node's data portion (in bytes). Only the "used" portions of the node's file blocks are counted; the amount of storage the node actually requires (i.e. the number of blocks the node consumes) may be larger than the size given here.

The size measurement *doesn't* include the node's attributes.

RETURN CODES

- `B_OK`. Success.
- `B_NO_MEMORY`. Couldn't get the necessary resources to complete the transaction.
- `B_BAD_VALUE`. The node doesn't exist (abstract entry).

GetStat()

virtual status_t **GetStat(**struct stat *_st_**)** const

`GetStat()` returns the **stat** structure for the node. The structure is copied into the *st* argument, which must be allocated. The **stat** structure is described in "The stat Structure", below. The BStatable object does *not* cache the **stat** structure; every time you call `GetStat()`, fresh **stat** information is retrieved.

RETURN CODES

- `B_OK`. Success.
- `B_NO_MEMORY`. Couldn't get the necessary resources to complete the transaction.
- `B_BAD_VALUE`. The node doesn't exist (abstract entry).

IsDirectory() see IsFile()

IsFile() , IsDirectory() , IsSymLink()

bool **IsFile(**void**)** const
bool **IsDirectory(**void**)** const
bool **IsSymLink(**void**)** const

These functions return **true** if the node is a plain file, a directory, or a symbolic link, respectively.

IsSymLink() see IsFile()
SetAccessTime() see GetCreationTime()
SetCreationTime() see GetCreationTime()
SetGroup() see GetOwner()
SetModificationTime() see GetCreationTime()
SetOwner() see GetOwner()
SetPermissions() see GetOwner()

The stat Structure

Declared in: be/posix/sys/stat.h

The `stat` structure looks like this:

```
struct stat {
   dev_t         st_dev;
   ino_t         st_ino;
   mode_t         st_mode;
   nlink_t         st_nlink;
   uid_t         st_uid;
   gid_t         st_gid;
   off_t         st_size;
   dev_t         st_rdev;
   size_t         st_blksize;
   time_t         st_atime;
   time_t         st_mtime;
   time_t         st_ctime;
} stat;
```

And the fields are...

- **st_dev** identifies the node's device.
- **st_ino** is the node's "inode" number.

By combining **st_dev** and **st_ino** you can roll your own **node_ref**:

```
node_ref nref;
stat st;

if (file.GetStat(&st) == B_OK) {
   nref.dev = st.st_dev;
   nref.node = st.st_ino;
}
```

Meanwhile...

- **st_mode** describes the node's flavor: plain file, directory or symbolic link. To test the field, pass it to the **S_ISREG()**, **S_ISDIR()**, and **S_ISLNK()** boolean macros:

```
if (S_ISREG(st.st_mode))
   /* it's a "regular" file */
   else if (S_ISDIR(st.st_mode))
      /* it's a directory */
```

```
    else if (S_ISLNK(st.st_mode))
       /* it's a symbolic link */
```

- **st_nlink** is the number of "hard links" that point to this node.
- **st_uid** and **st_gid** are the user (owner) and group ids that were described in the GetOwner() function.
- **st_rdev** is, well, no one really knows. It's provided for System V compatibility (hold your applause), but it's unused.
- **st_blksize** is the "preferred block size" that's used during copying. The **cp** command line program allocates buffers of this size when it's copying the file's data.
- **st_atime**, **st_mtime**, and **st_ctime** are the access, modification, and creation times in seconds since January 1, 1970. Access time (**st_atime**) is currently unused.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStatable Function Summary

Function Summary

Function	Description
GetCreationTime()	Returns the node's creation time
GetGroup()	Returns the node's permission bits
GetModificationTime()	Returns the node's last access time
GetNodeRef()	Returns the node's node_ref structure
GetOwner()	Returns the node's owner id
GetSize()	Returns the size of the node's data portion
GetStat()	Returns the stat structure for the node
SetCreationTime()	Returns the node's last modification time
SetOwner()	Returns the node's group id

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BVolume

Derived from: *none*
Declared in: be/storage/Volume.h
Library: libbe.so
Summary: *more...*

The BVolume class lets you ask questions about specific "volumes," where a volume is any independent file system. Most applications are usually only interested in "persistent" volumes, such as hard disks, floppies, or CD-ROMs, but you can also create BVolumes to virtual file systems, such as **/pipe**. Here's what a BVolume knows:

- The volume's name, device ID, and "root directory."
- Its storage capacity, and the currently available storage.
- If the volume is on a media that's removable.
- If the volume's storage is persistent (as opposed to the ephemeral storage you get with virtual file systems).
- If the volume is accessed through the network.
- If the file system uses MIME as file types, if it responds to queries, and if it knows about attributes.

Initializing a BVolume

There are two ways to initialize a BVolume:

- You can initialize it directly using a device ID (**`dev_t`**) that you pass to the BVolume constructor or **`SetTo()`** function. You can get a device ID from the **`device`** field of an **`entry_ref`** or **`node_ref`** structure. This method is useful if you have a file and you want to know which volume it lives on.
- If you want to iterate over all the mounted volumes, you can ask a BVolumeRoster object to get you the "next" volume (**`BVolumeRoster::GetNextVolume()`**). You can also ask the BVolumeRoster for the "boot" volume. This is the volume that was used to boot the computer.

Mount and Unmount

A BVolume object can't tell you directly whether the device that it represents is still mounted. If you want to ask, you can call a **`status_t`**-returning BVolume function; if the function returns **`B_BAD_VALUE`**, the device is no longer mounted.

Furthermore, you can't ask a BVolume to unmount itself. If you want to be told when devices are mounted and unmounted, you have to ask the Node Monitor to help you. Call **watch_node()** thus:

```
watch_node(NULL, B_WATCH_MOUNT, messenger);
```

messenger is a BMessenger object that acts as the target of subsequent mount and unmount notifications. See "The Node Monitor" section of this chapter for details.

Constructor and Destructor

BVolume()

```
BVolume(void)
BVolume(BVolume &volume)
BVolume(dev_t device)
```

Creates a new BVolume object and initializes it according to the argument. The status of the initialization is recorded by the `InitCheck()` function.

- The default constructor does nothing and sets `InitCheck()` to `B_NO_INIT`.
- The *device* constructor sets the BVolume to point to the volume represented by the argument. See the `SetTo()` function for status codes.
- The copy constructor sets the object to point to the same device as does the argument.

~BVolume

```
~BVolume()
```

Destroys the BVolume object.

Member Functions

Capacity() , FreeBytes()

```
off_t Capacity(void) const
off_t FreeBytes(void) const
```

Returns the volume's total storage capacity and the amount of storage that's currently unused. Both measurements are in bytes.

Device()

```
dev_t Device(void) const
```

Returns the object's **dev_t** number.

GetIcon()

```
status_t GetIcon(BBitmap *icon, icon_size which) const
```

Returns the volume's icon in *icon*. *which* specifies the icon to retrieve, either **B_MINI_ICON** (16x16) or **B_LARGE_ICON** (32x32).

See also: **get_device_icon()**

GetName()

```
status_t GetName(char *buffer) const
```

Copies the name of the volume into *buffer*.

GetRootDirectory()

```
status_t GetRootDirectory(BDirectory *dir) const
```

Initializes dir (which must be allocated) to refer to the volume's "root directory." The root directory stands at the "root" of the volume's file hierarchy. Note that this isn't necessarily the root of the entire file hierarchy.

InitCheck()

status_t InitCheck(void) const

Returns the status of the last initialization (from either the constructor or SetTo()).

IsPersistent() , IsRemovable() , IsReadOnly() , IsShared()

bool **IsPersistent**(void) const
bool **IsRemovable**(void) const
bool **IsReadOnly**(void) const
bool **IsShared**(void) const

These functions answer media-related questions about the volume:

- IsPersistent(). Is the storage persistent (such as on a floppy or hard disk)?
- IsRemovable(). Can the media be removed?
- IsReadOnly(). Can it be read but not written to?
- IsShared(). Is it accessed through the network (as opposed to being directly connected to this computer)?

KnowsAttr() , KnowsMime() , KnowsQuery()

bool **KnowsAttr**(void) const
bool **KnowsMime**(void) const
bool **KnowsQuery**(void) const

These functions answer questions about the file system on the volume:

- KnowsAttr(). Do its files accept attributes?
- KnowsMime(). Does it use MIME to type files?
- KnowsQuery(). Can it respond to queries?

SetTo() , Unset()

status_t SetTo(**dev_t** *dev*)
void **Unset**(void)

SetTo() initializes the BVolume object to represent the volume (device) identified by the argument.

Unset() uninitializes the BVolume.

Operators

= (assignment)

```
BVolume& operator = (const BEntry &volume)
```

In the expression

```
BVolume a = b;
```

BVolume *a* is initialized to refer to the same volume as *b*. To gauge the success of the assignment, you should call **InitCheck()** immediately afterwards. Assigning a BVolume to itself is safe.

Assigning from an uninitialized BVolume is "successful": The assigned-to BVolume will also be uninitialized (**B_NO_INIT**).

== , != (comparison)

```
bool operator == (const BVolume &volume) const
bool operator != (const BVolume &volume) const
```

Two BVolume objects are said to be equal if they refer to the same volume, or if they're both uninitialized.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BVolume Function Summary

Function Summary

Function	Description
!=	Tests if two objects refer to different volumes
=	Initializes the object to refer to the same volume as another
==	Tests if two objects refer to the same volume
Capacity()	Returns volume's total storage capacity
Device()	Returns volume's dev_t number
FreeBytes()	Returns free storage capacity for a volume
GetIcon()	Returns volume's icon
GetName()	Returns volume's name
GetRootDirectory()	Initializes a directory to refer to the volume's root directory
InitCheck()	Returns the status of the last initialization
IsPersistent()	Tests if the volume is persistent
IsReadOnly()	Tests if the volume is read only
IsRemovable()	Tests if the volume is removable
IsShared()	Tests if the volume is accessed through the network
KnowsAttr()	Tests if the volume's files accept attributes
KnowsMime()	Tests if the volume uses MIME types

KnowsQuery()	Tests if the volume responds to queries
SetTo()	Initializes the object to another volume
Unset()	Uninitializes the object
BVolume()	Constructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Constants and Defined Types

This section lists parts of the Storage Kit that aren't contained in classes.

Constants

Limits Constants

Declared in: be/storage/StorageDefs.h

Constant	Meaning
B_FILE_NAME_LENGTH	Number of characters allowed in a file name.
B_PATH_NAME_LENGTH	Number of characters allowed in a path name.
B_ATTR_NAME_LENGTH	Number of characters allowed in an attribute name.
B_MIME_TYPE_LENGTH	Number of characters allowed in a MIME type name.
B_MAX_SYMLINKS	Number of nested symlinks allowed.

These constants define the maximum values for several Storage Kit related items, including file and path name strings, attribute name strings, and MIME type strings. **B_MAX_SYMLINKS** specifies how many symbolic links may be linked through each other.

File Open Mode Constants

Declared in: be/storage/StorageDefs.h

Constant	Meaning
B_READ_ONLY	Open the file with read-only access.
B_WRITE_ONLY	Open the file with write-only access.
B_READ_WRITE	Open the file for both reading and writing.
B_FAIL_IF_EXISTS	Don't open the file if it already exists.
B_CREATE_FILE	Create the file before opening it.
B_ERASE_FILE	Erase the previous contents before opening the file.
B_OPEN_AT_END	Open with the pointer at the end of the file.

These constants are used when opening files using either the POSIX **open()** function or using the BFile class. They specify the mode in which the file is to be opened. For instance, if write-only access is desired, and you want the operation to fail if the file exists, you would do the following:

```
fd = open("foobar.data", B_WRITE_ONLY | B_FAIL_IF_EXISTS);
```

Node Flavors

Declared in: be/storage/StorageDefs.h

Constant	Meaning
B_FILE_NODE	Files only.

B_SYMLINK_NODE	Symbolic links only.
B_DIRECTORY_NODE	Directories only.
B_ANY_NODE	Matches any node.

These constants are used when making a request that can be qualified based on the "flavor" of a node—in other words, whenever you wish to perform an operation on only files, directories, or symbolic links. This is used when opening a file panel, for instance, so that you can specify what types of items you want the user to be able to select.

Version Kinds

Declared in: be/storage/AppFileInfo.h

Constant	Meaning
B_APP_VERSION_KIND	Records information about a specific application.
B_SYSTEM_VERSION_KIND	Records information about a "suite," or other grouping of applications, tha application belongs to.

These constants are used when setting or retrieving the version information attached to an application. There are two version information records for each application, and these two constants select which one you wish to reference. Although there is no prescribed use for these structures or their constants, it is suggested that **B_APP_VERSION_KIND** be used for application-specific version information, and **B_SYSTEM_VERSION_KIND** be used for information about the suite of applications to which the application belongs.

Directories

Declared in: be/storage/FindDirectory.h

Constant	Meaning
B_DESKTOP_DIRECTORY	Desktop directory.
B_TRASH_DIRECTORY	Trash directory.
B_APPS_DIRECTORY	Applications directory.
B_PREFERENCES_DIRECTORY	Preferences directory.
B_BEOS_DIRECTORY	BeOS directory.
B_BEOS_SYSTEM_DIRECTORY	System directory.
B_BEOS_ADDONS_DIRECTORY	BeOS add-ons directory.
B_BEOS_BOOT_DIRECTORY	Boot volume's root directory.
B_BEOS_FONTS_DIRECTORY	BeOS fonts directory.
B_BEOS_LIB_DIRECTORY	BeOS libraries directory.
B_BEOS_SERVERS_DIRECTORY	BeOS servers directory.
B_BEOS_APPS_DIRECTORY	BeOS applications directory.
B_BEOS_BIN_DIRECTORY	**/bin** directory.

B_BEOS_ETC_DIRECTORY	**/etc** directory.
B_BEOS_DOCUMENTATION_DIRECTORY	BeOS documentation directory.
B_BEOS_PREFERENCES_DIRECTORY	BeOS preferences directory.
B_COMMON_DIRECTORY	The common directory, shared by all users.
B_COMMON_SYSTEM_DIRECTORY	The shared system directory.
B_COMMON_ADDONS_DIRECTORY	The shared addons directory.
B_COMMON_BOOT_DIRECTORY	The shared boot directory.
B_COMMON_FONTS_DIRECTORY	The shared fonts directory.
B_COMMON_LIB_DIRECTORY	The shared libraries directory.
B_COMMON_SERVERS_DIRECTORY	The shared servers directory.
B_COMMON_BIN_DIRECTORY	The shared **/bin** directory.
B_COMMON_ETC_DIRECTORY	The shared **/etc** directory.
B_COMMON_DOCUMENTATION_DIRECTORY	The shared documentation directory.
B_COMMON_SETTINGS_DIRECTORY	The shared settings directory.
B_COMMON_DEVELOP_DIRECTORY	The shared develop directory.
B_COMMON_LOG_DIRECTORY	The shared log directory.
B_COMMON_SPOOL_DIRECTORY	The shared spool directory.
B_COMMON_TEMP_DIRECTORY	The shared temporary items directory.
B_COMMON_VAR_DIRECTORY	The shared **/var** directory.
B_USER_DIRECTORY	The user's home directory.
B_USER_CONFIG_DIRECTORY	The user's config directory.
B_USER_ADDONS_DIRECTORY	The user's add-ons directory.
B_USER_BOOT_DIRECTORY	The user's **/boot** directory.
B_USER_FONTS_DIRECTORY	The user's fonts directory.
B_USER_LIB_DIRECTORY	The user's libraries directory.
B_USER_SETTINGS_DIRECTORY	The user's settings directory.
B_USER_DESKBAR_DIRECTORY	The user's Deskbar directory.

These constants are used when calling the `find_directory()` function to determine the pathname of a particular directory of interest.

`B_DESKTOP_DIRECTORY` and `B_TRASH_DIRECTORY` are per-volume directories; if you don't specify the volume you wish to locate these directories on, `find_directory()` will assume you mean the boot disk.

`B_APPS_DIRECTORY` and `B_PREFERENCES_DIRECTORY` are global directories, and always refer to the standard **apps** and **preferences** directories.

The `B_BEOS_*` constants refer to BeOS-owned directories, the `B_COMMON_*` constants refer to directories that are common to all users of the system, and the `B_USER_*` constants refer to the current user's directories (currently these are all in a subtree rooted at **/boot/home**, but when

multiuser support is implemented in a future version of BeOS, these won't necessarily all be the same anymore).

In general, global application and system settings should be kept in `B_COMMON_*`; while settings that each user should be able to configure individually should be kept in `B_USER_*`.

By using these constants properly, your code will be compatible with future generations of the BeOS.

Icon Sizes

Declared in: be/storage/Mime.h

Constant	Meaning
`B_LARGE_ICON`	Large (32x32) icon.
`B_MINI_ICON`	Small (16x16) icon.

These constants are used when selecting icons from a meta MIME file; they let you fetch the large and small variations of a file's icon.

---

watch_node() Flags

Declared in: be/storage/NodeMonitor.h

Constant	Meaning
`B_STOP_WATCHING`	Stop watching the node.
`B_WATCH_NAME`	Watch for changes to the name of the node.
`B_WATCH_STAT`	Watch for stat changes.
`B_WATCH_ATTR`	Watch for attribute changes.
`B_WATCH_DIRECTORY`	Watch for changes to the directory's contents.
`B_WATCH_ALL`	Watch everything (except mounting).
`B_WATCH_MOUNT`	Watch for disk mounts and unmounts.

These constants are used to control what type of monitoring to perform on a node. `B_WATCH_ALL` is a convenience constant that allows you to monitor changes to the name, stat information, attributes, and directory of a node. `B_WATCH_MOUNT`, which is not included in `B_WATCH_ALL`, monitors volumes being mounted and unmounted.

`B_WATCH_DIRECTORY` applies only to directory nodes.

Node Monitor Opcodes

Declared in: be/storage/NodeMonitor.h

Constant	Meaning
`B_ENTRY_CREATED`	A new entry has been created.

B_ENTRY_REMOVED	An entry has been removed.
B_ENTRY_MOVED	An entry has been moved.
B_STAT_CHANGED	Stat information has changed.
B_ATTR_CHANGED	An attribute has been changed.
B_DEVICE_MOUNTED	A volume has been mounted.
B_DEVICE_UNMOUNTED	A volume has been unmounted.

These constants are returned in the "opcode" field of messages sent by the Node Monitor to let you know what sort of change has occurred to a node being monitored.

Query Operation Constants

Declared in: be/storage/Query.h

Constant	Operation
B_EQ	=
B_NE	!=
B_GT	>
B_LT	<
B_GE	>=
B_LE	<=
B_CONTAINS	string contains value ("**value**")
B_BEGINS_WITH	string begins with value ("*value**")
B_ENDS_WITH	string ends with value ("**value*")
B_AND	&&
B_OR	||
B_NOT	!

These constants define the operations that can be used to specify a query. They are used in conjunction with the push functions for constructing a query.

Defined Types

entry_ref

Declared in: be/storage/Entry.h

```
struct entry_ref {
        entry_ref();
        entry_ref(dev_t device, ino_t dir, const char *name);
        entry_ref(const entry_ref &ref);
        ~entry_ref();

      status_t     set_name(const char *name);
      bool      operator==(const entry_ref &ref) const;
      bool      operator!=(const entry_ref &ref) const;
      entry_ref &   operator=(const entry_ref &ref);

      dev_t        device;
      ino_t        directory;
      char        *name;
    }
```

The `entry_ref` structure describes a single entry in a directory.

`entry_ref()`
`entry_ref`(dev_t *device*, ino_t *dir*, const char **name*)
`entry_ref`(const entry_ref &*ref*)

- The constructor for the `entry_ref` structure. The first of these creates an empty `entry_ref`, the second accepts a device number, directory inode number, and a file name and constructs an `entry_ref` referring to that entry, and the last version of the constructor duplicates an existing `entry_ref`.

`~entry_ref()`

- The destructor for `entry_ref`.

status_t `set_name`(const char **name*)

- Lets you change the name of the file referred to by the **`entry_ref`** structure.

operator ==

- Lets you perform comparisons of `entry_ref` structures to see if they refer to the same entry.

operator !=

- Lets you test to see if two `entry_ref` structures refer to different entries.

device contains the device number on which the entry's target is located.

directory contains the inode of the directory that contains the entry's target.

name contains the name of the entry.

---

node_ref

Declared in: be/storage/Node.h

```
struct node_ref {
        node_ref();
        node_ref(const node_ref &ref);
        ~node_ref();

    bool    operator==(const node_ref &ref) const;
    bool    operator!=(const node_ref &ref) const;
    node_ref &  operator=(const node_ref &ref);

    dev_t       device;
    ino_t       node;
};
```

The `node_ref` structure describes a node in a file system.

`node_ref()`
`node_ref`(const node_ref &*ref*)

- The constructor for the `node_ref` structure. The first of these creates an empty `node_ref`, and the second duplicates an existing `node_ref`.

`~node_ref()`

- The destructor for `node_ref`.

operator ==

- Lets you perform comparisons of `node_ref` structures to see if they refer to the same node.

operator !=

- Lets you test to see if two `node_ref` structures refer to different nodes.

device contains the device number on which the node is located.

node contains the inode of the node.

---

version_info

Declared in: be/storage/AppFileInfo.h

```
struct version_info {
        uint32      major;
        uint32      middle;
        uint32      minor;
        uint32      variety;
        uint32      internal;
        char      short_info[64];
        char      long_info[256];
      }
```

The version_info structure is used to contain version information about an application. Although none of these fields have prescribed uses, and you can use them for anything you want, their names do hint at their suggested uses.

The Be Book The Storage Kit The Storage Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFile

Derived from: BNode, BPositionIO
Declared in: be/storage/File.h
Library: libbe.so
Summary: *more...*

A BFile lets you read and write the data portion of a file. It does this by implementing the `Read()` /`Write()` and `ReadAt()`/`WriteAt()` functions that are declared by the BPositionIO class.

Initializing and Opening

When you construct (or otherwise initialize) a BFile, the file is automatically opened. The file is closed when you re-initialize or destroy the object.

At each initialization, you're asked to supply an "open mode" value. this is a combination of flags that tells the object whether you want to read and/or write the file, create it if it doesn't exist, truncate it, and so on.

You can also initialize a BFile, and create a new file at the same time, through BDirectory's `CreateFile()` function. In this case, you don't have to supply an open mode—the BFile that's returned to you will automatically be open for reading and writing. (You *are* asked if you want the creation to fail if the named file already exists.)

Access to Directories and Symbolic Links

Although BFiles are meant to be used to access regular files, you aren't prevented from opening and reading a directory (you won't be able to write the directory, however). This isn't exactly a feature—there's not much reason to access a directory this way—you should simply be aware that it's not an error.

Symbolic links, however, *can't* be opened by a BFile—not because it's illegal, but because if you ask to open a symbolic link, the link is automatically traversed. The node that the BFile ends up opening will be the file or directory that the link points to.

This *is* a feature; very few applications should ever need to look at a symbolic link. (If yours is one of the few that *does* want to, you should go visit the BSymLink class.)

Constructor and Destructor

BFile()

```
BFile(void)
BFile(const BFile &file)
BFile(const entry_ref *ref, uint32 openMode)
BFile(const BEntry *entry, uint32 openMode)
BFile(const char *path, uint32 openMode)
BFile(BDirectory *dir, const char *path, uint32 openMode)
```

Creates a new BFile object, initializes it according to the arguments, and sets `InitCheck()` to return the status of the initialization.

The default constructor does nothing and sets `InitCheck()` to `B_NO_INIT`. To initialize the object, call `SetTo()`.

The copy constructor creates a new BFile that's open on the same file as that of the argument. Note that the two objects maintain *separate* data pointers into the *same* file:

- **Separate pointers:** Reading and writing through one object *does not* affect the position of the data pointer in the other object.
- **Same file:** If one object writes to the file, the other object will see the written data.

For information on the other constructors, see the analogous `SetTo()` functions.

~BFile()

```
virtual ~BFile()
```

Closes the object's file, frees its file descriptor, and destroys the object.

Member Functions

GetSize() , SetSize()

virtual status_t **GetSize(**off_t **size***)** const
virtual status_t **SetSize(**off_t *size***)**

These functions get and set the size, in bytes, of the object's file.

`GetSize()` returns the size of the file's data portion in the *size* argument; the measurement doesn't include attributes.

`SetSize()` sets the size of the data portion to the size given by the argument:

- Enlarging a file adds (uninitialized) bytes to its end.
- Shrinking a file removes bytes from the end.

RETURN CODES

- **`B_OK`**. The file's size was successfully gotten or set.
- **`B_NOT_ALLOWED`**. (`SetSize()`) The file lives on a read-only volume.
- **`B_DEVICE_FULL`**. (`SetSize()`) No more room on the file's device.

InitCheck()

status_t **InitCheck(**void**)** const

Returns the status of the most recent initialization.

RETURN CODES

- **`B_OK`**. The object is initialized.
- **`B_NO_INIT`**. The object is uninitialized.
- *See* **`SetTo()`** *for other errors.*

IsReadable() , IsWritable()

bool **IsReadable(**void**)** const
bool **IsWritable(**void**)** const

These functions tell you whether the BFile was initialized to read or write its file. If the object isn't (properly) initialized, they both return **false**.

Note that these functions don't query the actual file to check permissions, they only tell you what the access request was when the BFile object was initialized.

Position() see Seek()

Read() , ReadAt() , Write() , WriteAt()

virtual ssize_t **Read(**void *_buffer_, size_t _size_**)**
virtual ssize_t **ReadAt(**off_t _location_, void *_buffer_, size_t _size_**)**
virtual ssize_t **Write(**const void *_buffer_, size_t _size_**)**
virtual ssize_t **WriteAt(**off_t _location_, const void *_buffer_, size_t _size_**)**

These functions, which are inherited from BPositionIO, read and write the file's data; note that they don't touch the file's attributes.

The `Read()` and `ReadAt()` functions read *size* bytes of data from the file and place this data in *buffer*. The buffer that *buffer* points to must already be allocated, and must be large enough to accommodate the read data. Note that the read-into buffer is *not* null-terminated by the reading functions.

The two functions differ in that...

- `Read()` reads the data starting at the current location of the file's data pointer, and increments the file pointer as it reads.
- `ReadAt()` reads the data from the location specified by the *location* argument, which is taken as a measure in bytes from the beginning of the file. **`ReadAt()`** does *not* bump the file's data pointer.

`Write()` and `WriteAt()` write *size* bytes of data into the file; the data is taken from the *buffer* argument. The two functions differ in their use (or non-use) of the file's data pointer in the same manner as `Read()` and `ReadAt()`.

All four functions return the number of bytes that were actually read or written; negative return values indicate an error.

Reading fewer-than-*size* bytes isn't uncommon—consider the case where the file is smaller than the size of your buffer. If you want your buffer to be **NULL**-terminated, you can use the return value to set the **NULL**:

```
char buf[1024];
ssize_t amt_read;

if ((amt_read = file.Read((void *)buf, 1024)) < 0)
```

```
   /* handle errors first */
else
   /* otherwise set null */
   buf[amt_read] = '0';
```

A successful `Write()` or `WriteAt()`, on the other hand, will always write exactly the number of bytes you requested. In other words, **`Write()`** returns either the *size* value that you passed to it, or else it returns a negative (error) value.

Error codes returned by these functions can vary depending on the file system handling the operation. For this reason, specific error codes aren't listed here.

Seek() , Position()

virtual off_t **Seek(**off_t *offset*, int32 *seekMode***)**
virtual off_t **Position(**void**)** const

`Seek()` sets the location of the file's data pointer. The new location is reckoned as *offset* bytes from the position given by the *seekMode* constant:

Constant	Meaning
SEEK_SET	Seek from the beginning of the file.
SEEK_CUR	Seek from the pointer's current position.
SEEK_END	Seek from the end of the file.

If you `Seek()` to a position that's past the end of the file and then do a **`Write()`**, the file will be extended (padded with garbage) from the old end of file to the **`Seek()`**'d position. If you don't follow the `Seek()` with a **`Write()`**, the file isn't extended.

`Seek()` returns the new position as measured (in bytes) from the beginning of the file.

`Position()` returns the current position as measured (in bytes) from the beginning of the file. It doesn't move the pointer.

RETURN CODES

- **`B_ERROR`**. Attempted to `Seek()` "before" the beginning of the file, or you called `Position()` after such a `Seek()`. You also get **`B_ERROR`** if you call `Seek()` on an uninitialized file.
- **B_BAD_FILE**. `Position()` called on an uninitialized file.

If you do a "before the beginning" seek, subsequent **`Read()`** and **`Write()`** calls *do not fail*. But they almost certainly aren't doing what you want (you shouldn't be "before the file," anyway). The moral: Always check your **`Seek()`** return.

SetSize() see GetSize()

SetTo() , Unset()

```
status_t SetTo(const entry_ref *ref, uint32 openMode)
status_t SetTo(const BEntry *entry, uint32 openMode)
status_t SetTo(const char *path, uint32 openMode)
status_t SetTo(const BDirectory *dir, const char *path, uint32 openMode)
void Unset(void)
```

Closes the BFile's current file (if any), and opens the file specified by the arguments. If the specified file is a symbolic link, the link is automatically traversed (recursively, if necessary). Note that you're not prevented from opening a directory as a BFile, but you are prevented from writing it.

- In the *path* function, *path* can be absolute or relative, and can contain "." and ".." elements. If *path* is relative, it's reckoned off of the current working directory.
- In the *dir*/*path* function, *path* must be relative and is reckoned off of *dir*.

openMode is a combination of flags that determines how the file is opened and what this object can do with it once it is open. There are two sets of flags; you must pass one (and only one) of the following "read/write" constants:

Constant	Meaning
B_READ_ONLY	This object can read, but not write, the file.
B_WRITE_ONLY	This object can write, but not read, the file.
B_READ_WRITE	This object can read and write the file.

You can also pass any number of the following (these are optional):

Constant	Meaning
B_CREATE_FILE	Create the file if it doesn't already exist.
B_FAIL_IF_EXISTS	If the file already exists, the initialization (of the BFile object) fails.
B_ERASE_FILE	If the file already exists, erase all its data and attributes.
B_OPEN_AT_END	Sets the data pointer to point to the end of the file.

To open a file for reading and writing, for example, you simply pass:

```
file.SetTo(entry, B_READ_WRITE);
```

Here we create a new file or erase its data if it already exists:

```
file.SetTo(entry, B_READ_WRITE | B_CREATE_FILE | B_ERASE_FILE);
```

And here we create a new file, but only if it doesn't already exist:

```
file.SetTo(entry, B_READ_WRITE | B_CREATE_FILE | B_FAIL_IF_EXISTS);
```

Unset() closes the object's file and sets its InitCheck() value to B_NO_INIT.

RETURN CODES

- B_OK. The file was successfully opened.
- B_BAD_VALUE. **NULL** *path* in *dir*/*path*, or some other argument is uninitialized.
- B_ENTRY_NOT_FOUND. File not found, or couldn't create the file.
- B_FILE_EXISTS. File exists (and you set **B_FAIL_IF_EXISTS**).
- B_PERMISSION_DENIED. Read or write permission request denied.
- B_NO_MEMORY. Couldn't allocate necessary memory to complete the operation.

- Write() see Read()
- WriteAt() see Read()

Operators

= (assignment)

```
BFile& operator=(const BFile &File)
```

In the expression

```
BFile a = b;
```

BFile *a* is initialized to refer to the same file as *b*. To gauge the success of the assignment, you should call `InitCheck()` immediately afterwards. You can't assign a BFile to itself (`B_BAD_VALUE`).

Assigning to an uninitialized BFile is "successful": The assigned-to BFile will also be uninitialized (`B_NO_INIT`).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNode

Derived from: BStatable
Declared in: be/storage/Node.h
Library: libbe.so
Summary: *more...*

The BNode class gives you access to the data that a file system entry (a file, directory, or symbolic link) contains. There are two parts to this data:

- There's the "data portion" itself...
- ...and then there are the node's attributes.

The content of the data portion depends on the node's flavor:

- If it's a regular file, the data is whatever it is that the file is meant to contain: ASCII text, binary image or sound data, executable code, and so on. Note that resources (as created by the BResources class) are kept in the data portion.
- If it's a directory, the data is the list of entries that the directory contains.
- If it's a symbolic link, the data is the path of the "linked-to" file. The path can be absolute or relative.

The content of the attributes, on the other hand, isn't qualified by the node's flavor: Any node can contain any set of attributes.

Nodes are Dumb

Keep in mind that the concept of a "node" designates the *data parts* (data and attributes) of a file (a file, directory, or link). Contrast this with an "entry," which designates the entity's *location* within the file system: For example, you can write to a "node" (but not an entry), and you can rename an "entry" (but not a node).

This isn't just a conceptual crutch, it's the law: Nodes really don't know where they're located. For example, you can't ask a node for its name, or for the identity of its parent. This has some serious implications, the most important of which is...

- If you need to store a reference to a file (or directory, or symbolic link), don't store the node—in other words, don't cache the BNode object. Instead, store the information that you used to create the BNode (typically, a pathname or **`entry_ref`** structure).

Now that we've got that straight, we'll relax the rules a bit:

- BDirectory objects are node/entry hybrids. A BDirectory *does* know its own name (and parent, and so on).

This doesn't really change the "store the info" rule. Even if you're dealing exclusively with BDirectory objects, you should keep the generative information around. The primary reason for this is...

The "Node Pool" is Limited (File Descriptors)

Every BNode object consumes a "file descriptor." Your application can only maintain 256 file descriptors at a time. Because of this limit, you shouldn't keep BNodes around that you don't need. Keep in mind that BEntry objects also consumes file descriptors (one per object).

The file descriptor limit will probably be lifted, or at least settable, in a subsequent release. But even then you should be frugal.

Derived Classes and their Uses

BNode has three derived classes: BFile, BDirectory, and BSymLink. The derived classes define functions that let you access the node's data portion in the appropriate style; for example...

- BFile implements `Read()` and `Write()` functions that let you retrieve arbitrary amounts of data from arbitrary positions in the file.
- BDirectory implements functions, such as `GetNextEntry()` and `FindEntry()`, that read entries from the directory.
- BSymLink's `ReadLink()` returns the pathname that it contains.

If you want to (sensibly) look at a node's data portion, you must create an instance of the appropriate derived class. In other words, if you want to browse a directory, you have to create a BDirectory instance; if you want to write to a file, you create a BFile.

Be aware that it's not (always) an error to create an instance of the "wrong" derived class; setting a BFile to a symbolic link, for example, will traverse the link such that the BFile opens the file that the symbolic link is linked to. See the individual derived class specifications for more information.

BNode Instances

In practice, you almost always want to create an instance of one of the BNode-derived classes; but if, for whatever reason, you find yourself holding a BNode instance, here's what you'll be able to do with it:

- **Read and write attributes**. The attribute-accessing functions (`ReadAttr()`, `WriteAttr()`, and so on) are general—they work without regard for the node's flavor. Thus, you don't need an instance of a specific derived class to read and write attributes.
- **Get `stat` information.** The BStatable functions can be invoked on any flavor of node.
- **Lock the node**. This prevents other "agents" (other objects, other apps, the user) from accessing reading or writing the node's data and attributes. See "Node Locking".

Converting a BNode to an Instance of a Derived Class

This section describes situations and presents solutions to problems that are a bit esoteric. If you never create direct instances of BNode (and you never *have* to), then you should skip this and go to "Node Locking".

There may be times when you find yourself holding on to a BNode (instance) that you want to convert into a BFile, BDirectory, or BSymLink. However, you can't go *directly* from a BNode instance to an instance of BFile, BDirectory, or BSymLink—you can't tell your BNode to "cast itself" as one of its children.

There are solutions, however...

Converting to BDirectory

Converting from a BNode to a BDirectory, while not transparent, is pretty simple: Grab the **node_ref** out of the BNode and pass it to the BDirectory constructor or **SetTo()** function. Regard this example function:

```
void Node2Directory(BNode *node, BDirectory *dir)
{
   node_ref nref;

   if (!node || !dir) {
      dir.Unset();
      return;
   }

   node.GetNodeRef(&nref);

   /* Set the BDirectory. If nref isn't a directory node,
   * the SetTo() will fail.
   */
   dir.SetTo(&nref);
}
```

Converting to BFile or BSymLink

Converting a BNode instance to a BFile or BSymLink isn't as neat as the foregoing. Instead, you have to cache the information that you used to initialize the BNode in the first place, and then reuse it to create the BFile or BSymLink.

For example, let's say you receive an **entry_ref**. You turn it into a BNode, but then decide you need the data-writing power of a BFile. If, in the meantime, you lost the original **entry_ref**, you're sunk—there's nothing you can do.

Node Locking

Another feature provided by the BNode class is "node locking": Through BNode's **Lock()** function you can restrict access to the node. The lock is removed when **Unlock()** is called, or when the BNode object is deleted.

- When you lock a node, you prevent other objects (or agents) from reading or writing the node's data and attributes. No other agent can even *open* the node—other BNode constructions and POSIX open() calls (on that node) will fail while you hold the lock.
- You can only acquire a node lock if there are no file descriptors open on the node (with one exception). This means that no other BNode may be open on the node (locked or not), nor may the node be held open because of a POSIX `open()` (or `opendir()`) call.

The one exception to the no-file descriptors rule has to do with BEntries: Let's say you lock a directory, and then you initialize a BEntry to point to an entry within that directory. Even though the BEntry creates a file descriptor to the directory (as explained in the BEntry class), the initialization *will* succeed.

Implications

For files (and, less importantly, symlinks), the implications of locking are pretty clear: No one else can read or write the file. For directories, it's worth a closer look:

- Locking a directory means that the contents of the directory can't change: You can't create new nodes in the directory, or rename or remove existing ones. (You can, however, create abstract entries within the directory; see BEntry for more on abstract entries.)

Locking a node does *not* lock the node's entry: **You can't "lock out" entry operations, such as rename, move, and remove.** Even if you have a node locked, the entry that acts as the "container" for that node could disappear. If you want to prevent such operations on a node's entry, lock the entry's parent directory.

In general, you should try to avoid locking your nodes. If you must lock, try to make it brief. The primary reason (and, pretty much, the only reason) to lock is if separate elements in the data and/or attributes must be kept in a consistent state. In such a case, you should hold the lock just long enough to ensure consistency.

You shouldn't use locks to "privatize" data. Locking isn't meant to be used as a heightened permissions bit.

Constructor and Destructor

BNode()

BNode(const entry_ref *_ref_)
BNode(const BEntry *_entry_)
BNode(const char *_path_)
BNode(const BDirectory *_dir_, const char *_path_)

BNode()

BNode(const BNode &_node_)

Creates a new BNode object that's initialized to represent a specific file system node. To retrieve the status of the initialization, call **InitCheck()** immediately after constructing the object:

```
BNode node("/boot/lbj/FidoOnFire.gif");
if (node.InitCheck() != B_OK)
   /* The object wasn't initialized. */
```

A successfully initialized BNode object creates a "file descriptor" through which the object reads and writes the node's data and attributes. You can only have 256 file descriptors at a time (per application). The object's file descriptor is closed when the object is deleted, reset (through **SetTo()**), or unset (**Unset()**).

- **Default constructor**. The object's status will be **B_NO_INIT**, and the file descriptor isn't allocated until you actually initialize the object with a call to **SetTo()**.
- **Copy constructor**. The new BNode is set to the same node as the argument. Each of the two BNode objects has its own file descriptor.
- **Other constructors**. See the **SetTo()** functions.

~BNode()

virtual ~**BNode()**

Frees the object's file descriptor, unlocks the node (if it was locked), and destroys the object.

Member Functions

GetAttrInfo() , attr_info

status_t **GetAttrInfo**(const char *_attr_, attr_info *_info_) const
typedef struct **attr_info** {}

Gets information about the attribute named by *attr*. The information is copied into *info*, which must be allocated before it's passed in.

The `attr_info` structure, defined in **be/kernel/fs_attr.h**, is

```
typedef struct attr_info
{
   uint32 type;
   off_t size;
} attr_info;
```

- `type` is a constant (`B_STRING_TYPE`, `B_INT32_TYPE`, etc) that describes the type of data that the attribute holds.
- `size` is the size of the attribute's data, in bytes.

RETURN CODES

- `B_OK`. Success.
- `B_ENTRY_NOT_FOUND`. The node doesn't have an attribute named *attr*.
- `B_FILE_ERROR`. The object is uninitialized.

GetNextAttrName() , RewindAttrs()

status_t **GetNextAttrName**(char *_buffer_)
status_t **RewindAttrs**(void)

Every BNode maintains a pointer into its list of attributes. `GetNextAttrName()` retrieves the name of the attribute that the pointer is currently pointing to, and then bumps the pointer to the next attribute. The name is copied into the *buffer*, which should be at least **B_ATTR_NAME_LENGTH** characters long. The copied name is **NULL**-terminated. When you've asked for every name in the list, `GetNextAttrName()` returns an error.

`GetNextAttrName()` does *not* clear its argument if it returns an error. This will be corrected in a subsequent release.

`RewindAttrs()` resets the BNode's attribute pointer to the first element in the list.

To visit every attribute name, you would do something like this:

```
/* Print every attribute name. */
char buf[B_ATTR_NAME_LENGTH];

while (node.GetNextAttrName(buf) == B_OK) {
   printf("> Attr name: %sn", buf);
}
```

The attribute list is not static; when you ask for the next attribute name, you're asking for the next name in the list *as it exists right now*.

Furthermore, the ordinal position of an attribute within the list is indeterminate. "Newer" attributes are not necessarily added to the end of the list: If you alter the list while you're walking through it, you may get curious results—you may not see the attribute that you just now added (for example).

In general, it's best to avoid altering the list while you're iterating over it.

RETURN CODES

- `B_OK`. Success.
- `B_ENTRY_NOT_FOUND`. You've hit the end of the list.
- `B_FILE_ERROR`. The object is uninitialized.

InitCheck()

```
status_t InitCheck(void) const
```

Returns the status of the most recent initialization.

RETURN CODES

- `B_OK`. The object was successfully initialized.
- `B_NO_INIT`. The object is uninitialized.
- See the `SetTo()` function for a list other return values.

Lock() , Unlock()

```
status_t Lock(void)
status_t Unlock(void)
```

Locks and unlocks the BNode's node. While the node is locked, no other object can access the node's data or attributes. More precisely, no other agent can create a file descriptor to the node. If a file descriptor already exists to this node, the `Lock()` function fails.

See "Node Locking" for details.

RETURN CODES

- **`B_OK`**. The node was successfully locked or unlocked.
- **`B_BUSY`**. (`Lock()`) The node can't be locked.
- **`B_BAD_VALUE`**. (`Unlock()`) The node isn't locked.
- **`B_FILE_ERROR`**. The object is uninitialized.

ReadAttr() , WriteAttr() , RemoveAttr()

```
ssize_t ReadAttr(const char *name,
    type_code type,
    off_t offset,
    void *buffer,
    size_t length)
```

```
ssize_t WriteAttr(const char *name,
    type_code type,
    off_t offset,
    const void *buffer,
    size_t length)
```

```
status_t RemoveAttr(const char *attr)
```

These functions read, write, and remove the node's attributes. Attributes are name/data pairs, where names must be unique (within a given node) and the data can be of arbitrary length.

`ReadAttr()` reads the data in the attribute named *name*, and copies it in *buffer*. The length of the buffer (the maximum number of bytes to copy) is given by *length*. Currently, the *type* and *offset* arguments are unused (or unreliable). The function returns the number of bytes that were actually read.

`WriteAttr()` erases the data currently held by *name* (if such an attribute exists) and replaces it with a copy of the first *length* bytes of data in *buffer*. The *type* argument *is* remembered—you can retrieve an attribute's type through **`GetAttrInfo()`**, for example—and you need to specify the correct type when you're forming a query (see BQuery and the note below). But, as mentioned above, you don't need to match types when you're reading the attribute. The *offset* argument is currently unreliable and shouldn't be used. The functions returns the number of bytes that were written.

If you want to use the attribute in a query, its type must be either string, **`int32`**, **`uint32`**, **`int64`**, **`uint64`**, **`double`**, or **`float`**. (In other words, *type* must be **`B_STRING_TYPE`**, or **`B_INT32_TYPE`**, or

B_UINT32_TYPE, and so on.)

The value of an indexed attribute must be no more than 255 bytes long.

RemoveAttr() deletes the attribute given by *name*.

RETURN CODES

ReadAttr() and WriteAttr(), if successful, return the number of bytes read or written.

- B_OK. (Remove) The attribute was successfully removed.
- B_ENTRY_NOT_FOUND. (ReadAttr() and Remove()) The attribute doesn't exist.
- B_FILE_ERROR. The object is uninitialized.
- B_FILE_ERROR. (WriteAttr() and Remove()) This object is a read-only BFile.
- B_NOT_ALLOWED. (WriteAttr() and Remove()) The node is on a read-only volume.
- B_DEVICE_FULL. (WriteAttr()) Out of disk space.
- B_NO_MEMORY. (WriteAttr()) Not enough memory to complete the operation.

RemoveAttr() see ReadAttr()

RenameAttr()

status_t **RenameAttr(**const char ***name*, const char ***new_name***)**

Moves the attribute given by *name* to *new_name*. If *new_name* exists, it's clobbered.

RETURN CODES

- B_OK. The attribute was successfully renamed.
- B_ENTRY_NOT_FOUND. The *name* attribute doesn't exist.
- B_FILE_ERROR. The object is uninitialized.
- B_FILE_ERROR. This object is a read-only BFile.
- B_NOT_ALLOWED. The node is on a read-only volume.

RewindAttrs see GetNextAttrName()

SetTo() , Unset()

status_t **SetTo**(const entry_ref *_ref_)
status_t **SetTo**(const BEntry *_entry_)
status_t **SetTo**(const char *_path_)
status_t **SetTo**(const BDirectory *_dir_, const char *_path_)

void **Unset**(void)

Closes the BNode's current file descriptor and opens it on the node (of the entry) that's designated by the arguments.

- In the *path* version, *path* can be absolute or relative, and can contain "." and ".." elements. If *path* is relative, it's reckoned off of the current working directory.
- In the *dir*/*path* version, *path* must be relative. It's reckoned off of the directory given by *dir*.

BNode instances never traverse symbolic links. If the designated entry is a symbolic link, the BNode will open the link's node. (Conversely, BFile instances *always* traverse symbolic links.)

`Unset()` closes the BNode's file descriptor and sets `InitCheck()` to `B_NO_INIT`.

RETURN CODES

- `B_OK`. All is well.
- `B_ENTRY_NOT_FOUND`. The designated entry doesn't exist.
- `B_BAD_VALUE`. Uninitialized or malformed argument.
- `B_BUSY`. The node is locked.

Sync()

status_t **Sync**(void)

Immediately performs any pending disk transactions for the file, returning `B_OK` on success and an appropriate error message otherwise.

Unlock() see Lock()
Unset() see SetTo()
WriteAttr() see ReadAttr()

Operators

= (assignment)

```
BNode& operator=(const BNode &node)
```

In the expression

```
BNode a = b;
```

BNode *a* is initialized to refer to the same node as *b*. To gauge the success of the assignment, you should call `InitCheck()` immediately afterwards. It's safe to assign a BNode to itself.

== , != (comparison)

```
bool operator==(const BNode &node) const
bool operator!=(const BNode &node) const
```

Two BNode objects are said to be equal if they're set to the same node, or if they're both `B_NO_INIT`.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNode Function Summary

Function Summary

Function	Description
!=	Tests if two objects refer to different nodes
=	Initializes the node to refer to the same node as another
==	Tests if two objects refer to the same node
GetAttrInfo()	Returns information about a given attribute
GetNextAttrName()	Iterates through the names of the node's attributes
InitCheck()	Returns the status of the last initialization
Lock()	Locks the node
BNode()	Constructor
~BNode()	Destructor
ReadAttr()	Reads an attribute
RemoveAttr()	Removes an attribute
RenameAttr()	Renames an attribute
RewindAttrs()	Resets the attribute iterator
SetTo()	Initializes the object to another node
Sync()	Immediately performs any pending disk transactions for the node
Unlock()	Unlocks the node

Unset()	Uninitializes the object
WriteAttr()	Writes an attribute

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BResources

Derived from: *none*
Declared in: be/storage/Resources.h
Library: libbe.so
Summary: *more...*

You may not want to be here... The BResources class was designed for a specific purpose: To provide a means to bundle application "resources" (icons, in particular) within the application executable itself. If you want to add new resources to your own application (resources that you want to have "stick" to the executable), then you've come to the right place. But you shouldn't use BResources to add data to a regular data file—use attributes instead.

The data that a file contains is either "flat," or it's "structured." To read a flat file, you simply open it (through a BFile object) and start `Read()`'ing. Structured data requires that you understand the structure. Typically, an application understands the structure either because it's a well-known format, or because the application itself wrote the file in the first place.

The BResources class defines a simple design for storing structured data. The structure is a series of "resources," where each resource is key/value pair. A single "resource file" can hold an unlimited number of resources; a single resource within a resource file can contain an unlimited amount of data.

Resources are sort of like attributes in that they store chunks of data that are looked up through the use of a key. But note these differences:

- Resources are stored in the file itself, such that if you copy the file, you copy the resources, as well.
- Resources can't be queried.
- Only plain files can have resources. (In other words, directories and symbolic links can't have resources.)

Initializing a BResources Object

The BResources class provides the means for reading and writing a file's resources, but it doesn't let you access the file directly. Instead, you must initialize the BResources object by passing it a valid BFile object, either in the constructor or the `SetTo()` function. Note the following:

- The BFile that you pass in is copied by the BResources object. Thus, initializing a BResources object opens a new file descriptor into the file. You can delete the "original" BFile immediately after you use it to initialize the BResources object.

- Care must be taken to avoid writing to a BFile that other applications have open for reading. BResources can't enforce this rule, but if you're not careful to abide by it, problems can (and will) occur. Likewise, multiple applications mustn't open the same file for writing at the same time.
- If you want to write resources, the BFile must *not* be locked when you pass it in. The BResources needs to be able to lock its copy of your object.
- The BFile *must* be open for reading (at least).
- Unfortunately, BResources lacks an `InitCheck()` function. If you want to check initialization errors, you should always initialize through **`SetTo()`**, rather than through the constructor.

Identifying and Creating Resource Files

You can't use just any old file as a BResources initializer: The file must be an actual resource file. Simply initializing a BResources object with an existing non-resource file will *not* transform the file into a resource file—unless you tell the initializer to clobber the existing file.

For example, this initialization fails:

```
/* "fido" exists, but isn't a resource file. */
BFile file("/boot/home/fido", B_READ_WRITE);
BResources res;
status_t err;

if ((err = res.SetTo(&file)) != B_OK)
...
```

And this one succeeds...

```
/* The second arg to SetTo() is the "clobber?" flag. */
if ((err = res.SetTo(&file, true)) != B_OK)
...
```

...but at a price: **fido**'s existing data is destroyed (truncated to 0 bytes), and a new "resource header" is written to the file. Having gained a resource header, **fido** can thereafter be used to initialize a BResources object.

Clobber-setting a resource file is possible, but, as mentioned at the top of this class description, you'll probably never create resource files directly yourself

So where do resource files come from if you don't create them yourself? Step right up...

Executables as Resource Files

The only files that are naturally resource-ful are application executables. For example, here we initialize a BResources object with the **IconWorld** executable:

```
BPath path;
BFile file;
BResources res;

find_directory(B_APPS_DIRECTORY, &path);
path.Append("IconWorld");
file.SetTo(&path, B_READ_ONLY);

if (res.SetTo(&file) != B_OK)
   ...
```

The BResources object is now primed to look at **IconWorld**'s resources. But be aware that an application's "app-like" resources (its icons, signature, app flags) should be accessed through the BAppFileInfo class.

Resource Data

After you've initialized your BResources object, you use the *Fiddle`Resource()`* functions to examine and manipulate the file's resources:

Generative Functions

- **`AddResource()`** adds a new resource to the file.
- **`RemoveResource()`** removes an existing resource from the file.

Data Function

- **`LoadResource()`** loads a resource from disk and returns a pointer to it.

Info Functions

- **`HasResource()`** tells you if the file contains a specified resource.
- **`GetResourceInfo()`** returns information about a resource.

As mentioned earlier, the BFile that you use to initialize a BResources object must be open for reading. If you also want to modify the resources (by adding, removing, or writing) the BFile must also be open for writing.

Identifying a Resource within a Resource File

A single resource within a resource file is tagged with a data type, an ID, and a name:

- The data type is one of the **`type_code`** types (**`B_INT32_TYPE`**, **`B_STRING_TYPE`**, and so on) that characterize different types of data. The data type that you assign to a resource doesn't restrict the type of data that the resource can contain, it simply serves as a way to label the type of data that you're putting into the resource so you'll know how to cast it when you retrieve it.

- The ID is an arbitrary integer that you invent yourself. It need only be meaningful to the application that uses the resource file.
- The name is optional, but can be useful: You can look up a resource by its name, if it has one.

Taken singly, none of these tags needs to be unique: Any number of resources (within the same file) can have the same data type, ID, or name. It's the *combination* of the data type constant and the ID that uniquely identifies a resource within a file. The name, on the other hand, is more of a convenience; it never needs to be unique when combined with the data type or with the ID.

Some functions also provide the option to use a pointer to a resource's data to identify the resource; once a resource has been loaded into memory by calling **`LoadResource()`**, you can use the resulting pointer to identify it.

Data Format

All resource data is assumed to be "raw": If you want to store a **`NULL`**-terminated string in a resource, for example, you have to write the **`NULL`** as part of the string data, or the application that reads the resource from the resource must apply the **`NULL`** itself. Put more generally, the data in a resource doesn't assume any particular structure or format, it's simply a vector of bytes.

Data Ownership

Resource data that you retrieve from a BResources object belongs to the BResources object. You mustn't **`free()`** these pointers.

Individual changes that you make to the resource file are cached in memory until you call the **`Sync()`** function. Other applications won't see the changes until then.

Reading and Writing a Resource File as a Plain File

Just because a file is a resource file, that doesn't mean that you're prevented from reading and writing it as a plain file (through the BFile object). For example, it's possible to create a resource file, add some resources to it, and then use a BFile object to seek to the end of the file and write some flat data. But you have to keep track of the "data map" yourself—if you go back and add more resources to the file (or extend the size of the existing ones), your flat data will be overwritten: The BResources object doesn't preserve non-resource data that lives in the file that it's operating on.

Constructor and Destructor

BResources()

```
BResources(void)
BResources(BFile *file, bool clobber = false)
```

Creates a new BResources object. You can initialize the object by passing a pointer to a valid BFile; without the argument, the object won't refer to a file until **SetTo()** is called.

If *clobber* is true, the file that's referred to by BFile is truncated (it's data is erased), and a new resource file header is written to the file. If *clobber* is false and the file doesn't otherwise doesn't have a resource header, the initialization fails.

BResources copies the BFile argument; after the constructor returns, you can, for example, delete the BFile that you passed in.

~BResources()

```
virtual ~BResources()
```

Destroys the BResources object.

Member Functions

AddResource()

```
status_t AddResource(type_code type,
    int32 id,
    const void *data,
    size_t length,
        const char *name = NULL)
```

Adds a new resource to the file. For this function to have an effect, the file must be open for writing. The arguments are:

- *type* is one of the `type_code` constants defined in **be/support/TypeConstants.h**.
- *id* is the ID number that you want to assign to the resource. The value of the ID has no meaning other than that which your application gives it; the only restriction on the ID is that the combination of it and the data type constant must be unique across all resources in this resource file.
- *data* is a pointer to the data that you want the resource to hold.
- *length* is the length of the *data* buffer, in bytes.
- *name* is optional, and needn't be unique. Or even interesting.

Ownership of the *data* pointer isn't assigned to the BResources object by this function; after `AddResource()` returns, your application can free or otherwise manipulate the buffer that *data* points to without affecting the data that was written to the file.

RETURN CODES

- `B_OK`. The resource was successfully added.
- `B_NOT_ALLOWED`. The is open read-only.
- `B_FILE_ERROR`. The file's resource map doesn't exist or is invalid.
- `B_BAD_VALUE`. The data pointer is `NULL`.
- `B_NO_MEMORY`. Not enough memory to add the resource to the file.

File()

```
const BFile &File(void) const
```

Returns the BFile the BResource object references.

GetResourceInfo()

```
bool GetResourceInfo(int32 byIndex,
    type_code *typeFound,
    int32 *idFound,
    const char **nameFound,
    size_t *lengthFound)
bool GetResourceInfo(type_code byType,
    int32 andIndex,
    int32 *idFound,
    const char **nameFound,
    size_t *lengthFound)
bool GetResourceInfo(type_code byType,
    int32 andId,
    const char **nameFound,
    size_t *lengthFound)
bool GetResourceInfo(type_code byType,
    const char *andName,
    int32 *idFound,
    size_t *lengthFound)
bool GetResourceInfo(const void *byPointer
    type_code *typeFound,
    int32 *idFound,
    size_t *lengthFound
    const char **nameFound)
```

These functions return information about a specific resource, as identified by the first one or two arguments:

- The first version (*byIndex*) searches for the *byIndex*'th resource in the file.
- The second (*byType*/*andIndex*) searches for the *byIndex*'th resource that has the given type.
- The third (*byType*/*andId*) looks for the resource with the unique combination of type and ID.
- The fourth (*byType*/*andName*) looks for the first resource that has the given type and name.
- The last (*byPointer*) returns information about the resource whose data is pointed to by *byPointer*. This can be used to trace a resource that's already been loaded with `LoadResource()` back to its information.

The other arguments return the other statistics about the resource (if found).

The pointer that's returned in **foundName* belongs to the BResources. Don't free it.

The functions return **true** if a resource was found, and **false** otherwise.

HasResource()

```
bool HasResource(type_code type, int32 id)
bool HasResource(type_code type, const char *name)
```

Returns **true** if the resource file contains a resource as identified by the arguments, otherwise it returns `false`.

Keep in mind that there may be more than one resource in the file with the same *name* and *type* combination. The *type* and *id* combo, on the other hand, is unique. See "Identifying a Resource within a Resource File."

LoadResource()

```
const void *LoadResource(type_code type, int32 id, size_t *outSize)
const void *LoadResource(type_code type, const char *name, size_t *outSize)
```

Loads the specified resource into memory. The resource can be identified by either type code and ID or by type code and name. See "Identifying a Resource within a Resource File."

The returned pointer belongs to the resource file; it's valid until the resource gets changed. If an error occurs while trying to load the resource, **NULL** is returned.

MergeFrom()

```
status_t MergeFrom(BFile *fromFile)
```

Copies all the resource from the file specified by *fromFile* into the file targeted by the BResources object. The original file isn't changed. You can do this to a file that's opened read-only, but the changes won't have any effect.

RETURN CODES

- B_OK. The resources were copied without error.
- **B_BAD_FILE**. The resource map is empty.
- **B_FILE_ERROR**. A file error occurred, or the resource map is nonexistent.
- **B_IO_ERROR**. An error occurred while writing the data.
- **B_ERROR**. Something else went wrong.

PreloadResourceType()

status_t **PreloadResourceType(**type_code *type* = 0**)**

If you know you're going to need to access all resources of a particular *type*, you can preload them all into memory in one shot using this function. If you specify a *type* of 0, all resources of all types are preloaded.

RETURN CODES

- B_OK. The resources were preloaded without error.
- **B_BAD_FILE**. The resource map is empty.
- Other values. The returned value is the negative of the number of errors that occurred while preloading resources; for example, if five errors occurred, the result is -5.

RemoveResource()

status_t **RemoveResource(**type_code *type*, int32 *id***)**
status_t **RemoveResource(**const void **resource***)**

Removes the resource identified by the arguments. See "Identifying a Resource within a Resource File."

RETURN CODES

- **B_OK**. The resource was removed.
- **B_FILE_ERROR**. The file's resource map doesn't exist.
- **B_NOT_ALLOWED**. The file is opened read-only.
- **B_BAD_VALUE**. Couldn't find the specified resource, or an error occurred trying to remove it (first form of `RemoveResource()`).
- **B_ERROR**. Couldn't find the specified resource, the pointer doesn't indicate a valid resource, or an error occurred while removing the resource (second form).

SetTo()

status_t **SetTo(**BFile **file*, bool *clobber* = *false***)**

Unlocks and closes the object's previous BFile, and re-initializes it to refer to a copy of the argument. If the new BFile is open for writing, the BResources' copy of the BFile is locked.

If *clobber* is **true**, the file that's referred to by BFile is truncated (it's data is erased), and a new resource file header is written to the file. If *clobber* is **false** and the file doesn't otherwise doesn't have a resource header, the initialization fails.

RETURN CODES

- **B_OK**. The resource was removed.
- **B_BAD_VALUE**. The argument BFile is invalid (uninitialized).
- **B_ERROR**. The BResources couldn't be initialized (for whatever reason).

Sync()

status_t **Sync**(void)

Updates all changed resources on disk. This actually rewrites the *entire* resource file, so be aware of this when designing your code. This is a very good reason not to use BResources for anything other than permanent, nonchanging application data, and only developer tools should write to resource files.

RETURN CODES

- B_OK. The resources were updated without error.
- **B_BAD_FILE**. The resource map is empty.
- **B_NOT_ALLOWED**. The file is opened read-only.
- **B_FILE_ERROR**. A file error occurred.
- **B_IO_ERROR**. An error occurred while writing the data.

WriteTo()

status_t **WriteTo**(BFile **newFile*)

Writes all the file's resources into a new file. After this function returns, the BResources object is targeted on the new BFile; all future operations will be done in the new file.

RETURN CODES

- B_OK. The resources were written without error.
- Other errors. File errors indicating problems copying the data.
-

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BResources Function Summary

Function Summary

Function	Description
AddResource()	Adds a new resource to the file
File()	Returns the BFile the BResources object references
GetResourceInfo()	Returns information about a given resource
HasResource()	Tests whether a file contains the specified resource
LoadResource()	Loads the specified resource into memory
MergeFrom()	Copies all resources from another file into this one
PreloadResourceType()	Preloads all resources of the specified type
RemoveResource()	Removes a resource
BResources()	Constructor
~BResources()	Destructor
SetTo()	Initializes the object to another application's resources
Sync()	Writes all changed resources back out to disk
WriteTo()	Writes the file's resources into a new file

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPath

Derived from: BFlattenable
Declared in: be/storage/Path.h
Library: libbe.so
more...

A BPath object represents an absolute pathname, and provides some simple path manipulation and querying functions. The primary features of the class are:

- **It allocates storage for you**. When you tell your BPath object which pathname you want it to represent, the object allocates storage for the pathname automatically. When you delete the object, the storage is freed.
- **It always represents an absolute path**. The pathname strings that you use to initialize a BPath can be relative, and they can include references to "." and "..". The BPath "normalizes" the passed-in strings to create an absolute pathname, as described in "Initializing and Normalizing".

BPaths are handy, but don't expect them to actually *do* very much: **A BPath is just a pathname.** It identifies the location of a file, but it can't manipulate the file, nor can it change the structure of the file system.

So what do you use BPaths for?

- **You can use your BPaths to initialize other, more powerful objects** (BEntry, BNode and its kids). See "Converting a BPath" on page131.
- **BPaths can be passed through BMessages**. To add a BPath to a BMessage, you have to flatten it first: BPath implements BFlattenable for exactly this reason. The receiver of the BMessage can resurrect the flattened object as a BPath object or as an `entry_ref` structure. See "Passing a BPath in a BMessage".
- **BPath objects are ideal for caching references to files**. BPaths don't consume much in the way of system resources—they don't contain file descriptors, for example. So they're great for keeping track of the files that your application is interested in.

In the way that they're used, BPaths and `entry_ref`s are nearly identical. In particular, `entry_ref`s can do all three of the things listed here. Whether you use BPaths (pathnames in general) or `entry_ref`s is largely a matter of taste.

Initializing and Normalizing

You initialize a BPath—in other words, you establish the path that the object represents —by passing a string (or two, or a BDirectory and a string) to the constructor or to the `SetTo()` function. Upon initialization, the BPath object concatenates the strings and then "normalizes"

the passed-in strings *if it has to* (this emphasis is important, as we'll see in a moment). The following elements trigger normalization:

- a relative pathname (after concatenation; e.g. "boot/lbj")
- the presence of "." or ".." ("/boot/lbj/../lbj/./fido")
- redundant slashes ("/boot//lbj")
- a trailing slash ("/boot/lbj/")

During normalization, BPath conjures up an absolute pathname in the form

- */dir1/dir2/.../dirN/leaf*

It does this by applying the following rules:

- relative pathnames are reckoned off of the current working directory
- "." is ignored (at the head of a path, it's taken as the cwd).
- ".." bumps up one directory level
- redundant slashes are coalesced
- a trailing slash is removed.

(The one exception to this final rule is "/" as a full pathname.)

There's a subtle side-effect that you get with normalization: **When you normalize a pathname, all the elements in the path up to but not including the leaf must exist.** In other words, a normalized BPath object gives you the *same* guarantee of existence as does an **entry_ref** structure. The subtlety, here, is that an unnormalized BPath *needn't exist at all*.

For example, here we create a BPath for a pathname that contains a non-existent directory:

```
/* We'll assume that "/abc/def/" doesn't exist. */
BPath path("/abc/def/ghi.jkl");

/* Nonetheless, the BPath is successfully initialized.
* The Path() function returns a pointer to the object's
* pathname string.
*/
printf("Path: %sn". path.Path());
```

On the command line we see:

```
$ Path: /abc/def/ghi.jkl
```

But if we tickle the normalization machine...

```
/* The redundant slash causes a normalization. */
BPath path("/abc/def//ghi.jkl");
```

....the object is invalid:

```
$ Path: (null)
```

Forcing Initialization

Both the constructor and the **SetTo()** function carry an optional argument that lets you force the passed-in path to be normalized:

```
/* The trailing bool forces normalization. */
BPath path("/abc/def/ghi.jkl", true);
printf("Path: %sn", path.Path());
```

In this case, the forced normalization nullifies the object:

```
$Path: (null)
```

Normalization by Default?

Since forcing normalization makes BPath's behaviour more consistent and reliable, why not always normalize? Because **normalization can be expensive.**

During normalization, the pathname is `stat`'d and prodded rather heavily. If you're planning on using your BPath's pathname to initialize a BEntry or BNode, this prodding will happen again. Rather than incur the expense twice, you may want to live with unnormalized BPath objects, and take the normalization hit during the subsequent initialization.

Other Normalization Details

- **You can't force the BPath constructor or SetTo() function to skip the normalization.** If the path *needs* to be normalized, it *will* be normalized.
- **BPath doesn't let you ask if its pathname was normalized.**

The BPath Calling Convention

BPath objects are passed back to you (by reference) by a number of Storage Kit functions. However, you shouldn't find any functions that ask for a BPath object. This is a convention of usage:

- If an API element returns a pathname to you, it does so in the form of a BPath. If it asks for a pathname from you (as an argument), it asks for a **`const char *`**.

As an example of a function that returns a BPath to you, recall BEntry's **GetPath()** function:

```
status_t BEntry::GetPath(BPath *path)
```

(As an aside, this is where the auto-allocation comes in handy—because BPath allocates the pathname storage for you, you don't have to mess around with ugly buffer and length arguments.)

On the other hand, BEntry's **SetTo()** takes a pathname as a **const char ***:

```
status_t BEntry::SetTo(const char *path)
```

If you've got a BPath loaded up with a pathname, you would call this function thus:

```
entry.SetTo(path.Path());
```

The constructors and **SetTo()** functions in (most of) the Storage Kit classes have **const char *** versions that can be called as shown here.

Passing a BPath in a BMessage

Let's say you've got a BPath object that you want to send to some other application. To do this, you have to add it to a BMessage object through the latter's **AddFlat()** function. As an inheritor from BFlattenable, the BPath knows how to flatten itself for just this purpose.

```
BMessage msg;
BPath path("/boot/lbj/fido");

/* The check here is important, as we'll describe
* in a moment.
*/
if (msg.AddFlat("pathname", &path) != B_OK)
   /* handle the error */
```

The receiver of the message can retrieve the pathname as a BPath object by calling **FindFlat()**:

```
void MyApp::MessageReceived(BMessage *msg)
{
   BPath path;

   if (msg->FindFlat("pathname", &path) != B_OK)
      /* handle the error */
   ...
}
```

Alternatively, the pathname can be retrieved as an **entry_ref** through **FindRef()**:

```
void MyApp::MessageReceived(BMessage *msg)
```

```
{
   entry_ref ref;

   if (msg->FindRef("pathname", &ref) != B_OK)
      /* handle the error */
   ...
}
```

If you want to skip all the conversion business and simply pass the pathname as a string, use `AddString()`. The receiver, of course, would have to call `FindString()` to retrieve your pathname string.

What's Really Going On

When you add a flattened BPath to a BMessage, the object's pathname is turned into an `entry_ref`. If the message receiver asks for a BPath (through `FindFlat()`), the `entry_ref` is turned back into a BPath object. Therefore, i**t's more efficient to retrieve a flattened BPath as an entry_ref than it is to unflatten it as a BPath object.**

The BPath to `entry_ref` conversion has another, more subtle implication: **Adding a BPath through AddFlat() performs an implicit normalization on the data that's added to the BMessage.**

If the normalization fails, the `AddFlat()` function returns an error and the data isn't added to the BMessage. The original BPath is untouched, regardless of the result of the normalization.

Converting a BPath

As mentioned earlier, most of the Storage Kit classes have constructors and `SetTo()` functions that accept **`const char *`** arguments. If you want to turn your BPath into a BFile (for example), you would do this (including error checks):

```
status_t err;

BFile file(path.Path());
err = InitCheck();
```

or

```
err = file.SetTo(path.Path());
```

To convert a BPath to an `entry_ref`, pass the pathname to the `get_ref_for_path()` function:

```
entry_ref ref;
status_t err;

err = get_ref_for_path(path.Path(), &ref);
```

For you Node Monitor users: You can't convert directly to a **node_ref** structure. The quickest way from here to there is:

```
node_ref nref;
status_t err;

/* We'll skip InitCheck() and catch errors in GetNodeRef(). */
BEntry entry(path.Path());
err = entry.GetNodeRef(&nref);
```

Immutability

Remember, a BPath represents a pathname, not a node. It isn't "updated" when the file system changes:

- A BPath's pathname string never changes behind your back, even if the entry that it originally pointed to is renamed, moved, or deleted.

For example:

```
BEntry entry;
BPath path;

/* Set a BPath, construct a BEntry from it, rename
* the entry, and then print the BPath's pathname.
*/
if (path.SetTo("/boot/lbj/fido") == B_OK)
   if (entry.SetTo(&path) == B_OK)
      if (entry.Rename("rover") == B_OK)
         printf("Pathname: %sn", path.Path());
```

We see...

```
$ Pathname: /boot/lbj/fido
```

...even though the entry that the BPath was constructed to represent has been renamed.

Constructor and Destructor

BPath()

```
BPath(const char *path, const char *leaf = NULL, bool normalize = false)
BPath(const BDirectory *dir, const char *leaf = NULL, bool normalize = false)
BPath(const BEntry *entry)
BPath(const entry_ref *ref)
BPath(void)
BPath(const BPath &path)
```

Creates a new BPath object that represents the path that's created from the arguments. See the analogous `SetTo()` functions for descriptions of the flavorful constructors.

- The default constructor does nothing; it should be followed by a call to `SetTo()`.
- The copy constructor makes a copy of the argument's pathname.

The constructor automatically allocates memory for the object's stored pathname. The memory is freed when the object is deleted.

To check to see if an initialization was successful, call `InitCheck()`.

~BPath

```
virtual ~BPath()
```

Frees the object's pathname storage and extinguishes the object.

Member Functions

Append()

```
status_t Append(const char *path, bool normalize = false)
```

Appends the pathname given by *path* to the object's current pathname. *path* must be relative. If normalize is **true**, the new pathname is normalized; otherwise, it's normalized only if necessary.

Note that this...

```
Append("subdir/file")
```

...is the same as (and is implemented as):

```
path.SetTo(path.Path(), "subdir/file");
```

The `Append()` return value is picked up from the `SetTo()` call.

RETURN CODES

- `B_OK`. Success.
- `B_BAD_VALUE`. *path* contained a leading "/", or `this` is uninitialized.
- See `SetTo()` for other return values.

GetParent()

```
status_t GetParent(BPath *path) const
```

Initializes the argument with the pathname to the parent directory of `this`. Destructive parenting is acceptable (sociologically, it's a given):

```
BPath path("/boot/lbj/fido");

path.GetParent(&path);
```

Other details...

- `GetParent()` makes a call to `SetTo()`, but it's guaranteed not to tickle the normalization machine.
- You can't get the parent of "/".

RETURN CODES

- `B_OK`. Hello, mother.
- `B_ENTRY_NOT_FOUND`. You tried to get the parent of "/".
- `B_BAD_VALUE`. *path* is **`NULL`**.
- `B_NO_MEMORY`. Couldn't allocate storage for the pathname.

If the initialization isn't successful, the argument's `InitCheck()` is set to `B_NO_INIT`.

InitCheck()

```
status_t InitCheck(void) const
```

Returns the status of the most recent construction or `SetTo()` call.

RETURN CODES

- `B_OK`. The initialization was successful.
- `B_NO_INIT`. The object is uninitialized (this includes `Unset()`).
- See `SetTo()` for other errors.

Path() , Leaf()

```
const char *Path(void) const
const char *Leaf(void) const
```

These functions return the object's full path and leaf name, respectively. For example:

```
BPath path("/boot/lbj/fido");
printf("Path: %sn", path.Path());
printf("Leaf: %sn", path.Leaf());
```

Produces...

```
$ Path: /boot/lbj/fido
$ Leaf: fido
```

In both cases, the returned pointers belong to the BPath object. When the BPath is deleted, the pointers go with it.

If the BPath isn't initialized, the functions return pointers to **`NULL`**.

SetTo() , Unset()

```
status_t SetTo(const char *path, const char *leaf = NULL, bool normalize = false)
status_t SetTo(const BDirectory *dir, const char *leaf = NULL,
    bool normalize = false)
status_t SetTo(const BEntry *entry)
```

```
void Unset(void)
```

The `SetTo()` function frees the pathname that the object currently holds, and re-initializes the object according to the arguments:

- The first version concatenates the *path* and *leaf* strings (interposing a "/" if necessary). If *path* is relative, the concatenated pathname is appended to the current working directory. Note that you don't have to split your pathname into two parts to call this constructor; the optional *leaf* argument is provided simply as a convenience.
- The second version performs a similar operation using the path of the BDirectory as the initial part of the pathname.
- The third version initilizes the object with the path and name of the *entry*.

Regarding the *leaf* argument:

- The *leaf* string can contain directories—it needn't be *just* a leaf name.
- However, *leaf* must be a relative pathname (it can't start with "/").

If set to **`true`**, the *normalize* argument tells the object to normalize the new pathname. By default (**`false`**), the pathname is normalized only if necessary. Note that the default *doesn't* mean that the object absolutely won't normalize, it just won't do it if it doesn't think it's necessary. See "Initializing and Normalizing" on page128 for the full story on normalizing a pathname, including the conditions that trigger default normalization. Normalizing has no meaning with the BEntry version of `SetTo()`.

Storage for the pathname is allocated by the BPath object and is freed when the object is deleted (or when you re-initialize through `SetTo()`). The *path* and *leaf* arguments are copied into the allocated storage.

Other details...

- Destructive setting is safe:

```
/* This works... */
path.SetTo(path.Path(), ...);
```

- Currently, `SetTo()` only checks pathname and filename length if it has to normalize.

`Unset()` frees the object's pathname storage and sets the `InitCheck()` value to `B_NO_INIT`.

RETURN CODES

- **B_OK**. Successful initialization.
- **B_BAD_VALUE**. *path* is **NULL**, *leaf* isn't relative (it starts with a "/"), or *dir* is uninitialized.
- **B_BAD_VALUE**. A directory in the path doesn't exist (normalization only).
- **B_NAME_TOO_LONG**. A pathname element is too long (normalization only).
- **B_NO_MEMORY**. Couldn't allocate storage for the pathname.

The return value is also recorded in **InitCheck()**.

BFlattenable Functions

The following functions are implemented in accordance with the rules set down by the BFlattenable class. You never need to invoke these functions directly; they're implemented so a BPath can added to a BMessage (see "Passing a BPath in a BMessage" on page130). But in case you're interested...

AllowsTypeCode()

virtual bool **AllowsTypeCode(**type_code *code***)** const

Returns **true** if *code* is **B_REF_TYPE**, and **false** otherwise.

Flatten()

virtual status_t **Flatten(**void **buffer*, ssize_t *size***)** const

Converts the object's pathname to an **entry_ref** and writes it into *buffer*. Currently, *size* is ignored.

RETURN CODES

- **B_OK**. Peachy.
- **B_NAME_TOO_LONG**. The pathname is too long (> 1024 characters).
- **B_ENTRY_NOT_FOUND**. A directory in the path doesn't exist.

FlattenedSize()

virtual ssize_t **FlattenedSize()** const

Returns the size of the **entry_ref** that represents the flattened pathname.

IsFixedSize()

virtual bool **IsFixedSize()** const

Returns **false**.

TypeCode()

virtual type_code **TypeCode()** const

Returns **B_REF_TYPE**.

Unflatten()

virtual status_t **Unflatten(**type_code *code*,
const void **buffer*,
ssize_t *size***)**

Initializes the BPath with the flattened **entry_ref** data that's found in *buffer*. The type code must be **B_REF_TYPE**.

RETURN CODES

- **B_OK**. Success.
- **B_BAD_VALUE**. Wrong type code (not **B_REF_TYPE**).
- **B_ENTRY_NOT_FOUND**. A directory in the **entry_ref** data doesn't exist.

The **Unflatten()** return value is recorded in **InitCheck()**.

Operators

= (assignment)

```
BPath& operator=(const BPath &path)
BPath& operator=(const char *string)
```

Initializes **this** with a copy of the pathname that's gotten from the argument. Also sets **InitCheck()**.

== , != (comparison)

```
bool operator==(const BPath &path) const
bool operator==(const char *string) const
```

```
bool operator!=(const BPath &path) const
bool operator!=(const char *string) const
```

Compares **this**'s pathname with the pathname taken from the argument. The comparison is a simple **strcmp()**; neither path is normalized or otherwise altered before the comparison is made. For example:

```
BPath path("/boot/lbj/fido");

chdir("/boot");
printf("Are they equal? %dn", path == "lbj/fido");
```

Displays:

```
$ Are they equal? 0
```


The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPath Function Summary

Function Summary

Function	Description
!=	Tests if two objects represent different paths
=	Initializes the object with a copy of the path from the other
==	Tests if two objects represent the same path
AllowsTypeCode()	Provided for implementation
Append()	Appends a pathname to the current pathname
Flatten()	Provided for implementation
FlattenedSize()	Provided for implementation
GetParent()	Returns a BPath with a pathname to the parent directory
InitCheck()	Returns the status of the last initialization
IsFixedSize()	Provided for implementation
Leaf()	Returns the object's leaf name
Path()	Returns the object's full name
SetTo()	Initializes the object to another path
TypeCode()	Provided for implementation
Unflatten()	Provided for implementation
Unset()	Uninitializes the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDirectory

Derived from: BNode, BEntryList
Declared in: be/storage/Directory.h
Library: libbe.so
Summary: *more...*

A BDirectory object gives you access to the contents of a directory. A BDirectory's primary features are:

- **It can iteratively retrieve the entries in the directory**. The entries are returned as BEntry objects, **entry_refs**, or **dirent** structures (**GetNextEntry()**, **GetNextRef()**, **GetNextDirents()**).
- **It can find a specific entry**. You can ask if the entry exists (**Contains()**), and you can retrieve the entry as a BEntry (**FindEntry()**).
- **It can create new entries**. Through the aptly named **CreateFile()**, **CreateDirectory()** and **CreateSymLink()** functions.

Unlike the other BNode classes, a BDirectory knows its own entry (**GetEntry()**), and can be initialized with a **node_ref** structure.

Retrieving Entries

The BDirectory functions that let you iterate over a directory's entries are inherited from BEntryList:

- status_t GetNextEntry(BEntry **entry*, bool *traverse* = **true**);
  status_t GetNextRef(entry_ref **ref*);
  int32 GetNextDirents(dirent **buf*, size_t *length*, int32 *count* = INT_MAX)

For the basic story on these functions, see the BEntryList class and the function descriptions below. In addition to the info you'll find there, you should be aware of the following:

- **Entries are returned in "directory order"**. This is, roughly, the ASCII order of their names.
- **Try not to alter the directory while you're getting its entries.** Entries are delivered on demand. If you do something to change the contents of the directory while you're iterating through those contents (such as change the name of the file "aaa" to "zzz") you could end up seeing an entry more than once (technically, you'll see the same node under the guise of different entries), or you could miss an entry.
- **Counting entries uses the same iterator that retrieves entries**. You mustn't call **CountEntries()** while you're looping over a GetNext...() function.

Creating New Directories

To create a new directory, you can use BDirectory's `CreateDirectory()` function. The function creates a single new directory as identified by its argument. The new directory will be a subdirectory of the invoked-upon BDirectory's directory.

You can also create an entire path full of new directories through the global `create_directory()` function. This convenient function attempts to create *all* "missing" directories along the path that you pass in.

Finding a Directory

The `find_directory()` function gives you the pathnames for pre-defined directories. These directories, such as those that store Be-supplied applications and user-defined preferences settings, are represented by **`directory_which`** constants. These constants are not strings; you can't use them directly. You have to pass them through `find_directory()`.

Note that the BDirectory class itself doesn't let you find directories on the basis of the **`directory_which`** constants—you have to use the `find_directory()` function (which is documented at the end of this class description).

Node Monitoring a Directory

The following description is a brief, directory-specific view into the Node Monitor. For the full story, see "The Node Monitor" section of this chapter.

You can monitor changes to the contents of a directory by passing a BDirectory's `node_ref` and the **`B_WATCH_DIRECTORY`** flag to the Node Monitor's `watch_node()` function. As with all invocations of `watch_node()`, you also have to pass a BMessenger (the "target") that will receive the Node Monitor notifications; here, we use `be_app_messenger`:

```
BDirectory dir("/boot/home");
node_ref nref;
status_t err;

if (dir.InitCheck() == B_OK) {
   dir.GetNodeRef(&nref);
   err = watch_node(&nref, B_WATCH_DIRECTORY, be_app_messenger);
   if (err != B_OK)
      /* handle the error */
}
```

The following changes to the monitored directory cause BMessages to be sent to the target. The `what` field for all Node Monitor messages is `B_NODE_MONITOR`; the "opcode" field (an integer code) describes the activity:

- An entry was created (opcode = `B_ENTRY_CREATED`).
- An entry was moved to a different name in the same directory (**`B_ENTRY_RENAMED`**).

- An entry was from moved from this directory to a different directory, or vice versa (**B_ENTRY_MOVED**).
- An entry (and the node it represents) was deleted from the file system (**B_ENTRY_REMOVED**).

The **B_WATCH_DIRECTORY** flag (by itself) *doesn't* monitor changes to the directory's own entry. For example, if you change the name of the directory that you're monitoring, the target isn't sent a message. If you want a BDirectory to watch changes to itself, you have to throw in one of the other Node Monitor flags (**B_WATCH_NAME**, **B_WATCH_STAT**, or **B_WATCH_ATTR**).

The other fields in the Node Monitor message describe the entry that changed. The set of fields depends on the opcode (the following is a summary of the list given in "Notification Messages" in the Node Monitor documentation):

B_ENTRY_CREATED

Field	Type	Description
"device"	B_INT32_TYPE	dev_t of the directory's device.
"directory"	B_INT64_TYPE	ino_t (node number) of the directory.
"node"	B_INT64_TYPE	ino_t of the new entry's node.
"name"	B_STRING_TYPE	The name of the new entry.

B_ENTRY_MOVED

The "device", "node", and "name" fields are the same as for B_ENTRY_CREATED, plus...

Field	Type	Description
"from_directory"	B_INT64_TYPE	The ino_t number of the old directory.
"to_directory"	B_INT64_TYPE	The ino_t number of the new directory.

B_ENTRY_REMOVED

The B_ENTRY_REMOVED message takes the same form as B_ENTRY_CREATED, but without the "name" field. This, obviously, can be a problem—what good is it if you're told that a file has been removed, but you're not told the file's name? In some cases, simply being told that a file has been removed actually *is* good enough: You can simply re-read the contents of the directory.

Constructor and Destructor

BDirectory()

```
BDirectory(const entry_ref *ref)
BDirectory(const node_ref *nref)
BDirectory(const BEntry *entry)
BDirectory(const char *path)
BDirectory(const BDirectory *dir, const char *path)
BDirectory(void)
BDirectory(const BDirectory &directory)
```

Creates a new BDirectory object that represents the directory as given by the arguments. See the analogous SetTo() functions for descriptions of the flavorful constructors.

- The default constructor does nothing; it should be followed by a call to SetTo().
- The copy constructor points the BDirectory to the same directory as is represented by the argument. The two objects have their own entry iterators.

To check to see if an initialization was successful, call InitCheck().

~BDirectory()

```
virtual ~BDirectory()
```

Deletes the object.

Member Functions

Contains()

```
bool Contains(const char *path, int32 nodeFlags = B_ANY_NODE) const
bool Contains(const BEntry *entry, int32 nodeFlags = B_ANY_NODE) const
```

Returns true if *path* or *entry* is contained within this directory, or in any of its subdirectories (no matter how deep). You can use the *nodeFlags* argument to limit the search to a particular flavor of node:

- B_FILE_NODE looks for a "plain" file.
- B_DIRECTORY_NODE looks for a directory.
- B_SYMLINK_NODE looks for a symbolic link.
- B_ANY_NODE (the default) doesn't discriminate between flavors.

CountEntries() see GetNextEntry()

CreateDirectory() see CreateFile()

CreateFile() , CreateDirectory() , CreateSymLink()

```
status_t CreateFile(const char *path, BFile *file, bool failIfExists = false)
status_t CreateDirectory(const char *path, BDirectory *dir)
status_t CreateSymLink(const char *path, const char *linkToPath, BSymLink *link)
```

These functions create a new file, directory, or symbolic link. The new node is located at *path*. If *path* is relative, it's reckoned off of the directory represented by this BDirectory; if it's absolute, the path of this BDirectory is ignored.

- CreateFile() fails if the file already exists and *failIfExists* is true. If the flag is false (and the file exists), the old file is clobbered and a new one is created. If successful, the BFile argument that you pass in is opened on the new file in B_READ_WRITE mode.
- CreateDirectory() and CreateSymLink() fail if *path* already exists—you can't clobber an existing directory or link.
- The *linkToPath* argument (CreateSymLink()) is the path that the new symbolic link will be linked to.

The object argument (the BDirectory, BFile, or BSymLink) may be NULL. If the function fails, the object argument, if non-NULL, is Unset().

RETURN CODES

- B_OK. Success.
- B_BAD_VALUE. Illegal *path*, *file*, *dir*, or *link* specified; may be NULL. *path* may be empty.
- B_BUSY. A busy node could not be accessed.
- B_ENTRY_NOT_FOUND. The specified *path* does not exist or is an empty string.

- B_FILE_ERROR. A file system error prevented the operation.
- B_FILE_EXISTS. The file specified by *path* already exists.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_NAME_TOO_LONG. The *path* specified is too long.
- B_NO_MEMORY. Insufficient memory to perform the operation.
- B_NO_MORE_FDS. All file descriptors are in use (too many open files).
- B_IS_A_DIRECTORY. Can't replace a directory with a file.
- B_NOT_A_DIRECTORY. A component of the *path* is not a directory.
- B_NOT_ALLOWED. The volume is read-only.
- B_PERMISSION_DENIED. Create access is denied in the specified *path*.
- E2BIG. *linkToPath* is too long (CreateSymLink() only).

CreateSymLink() see CreateFile()

FindEntry()

```
status_t FindEntry(const char *path, BEntry *entry, bool traverse = false) const
```

Finds the entry with the given name, and sets the second argument to refer to that entry.

- *path* must be a relative pathname. It's reckoned off of the BDirectory's directory.
- You *are* allowed to look for "." and "..". The former represents this directory's entry. The latter refers to this directory's parent.
- The *entry* argument must be allocated before it's passed in (it needn't be initialized).
- The *traverse* applies to symbolic links: If the flag is true, the link is traversed. If it's false, you get the BEntry that points to the link itself.

If *path* isn't found, the second argument is automatically Unset(). To find out why the lookup failed, invoke InitCheck() on the *entry* argument:

```
BEntry entry;
status_t err;

if (dir.FindEntry("aFile", &entry) != B_OK) {
   err = entry.InitCheck();
}
```

The direct return value is also informative, but it may not be as precise as the InitCheck() value.

RETURN CODES

- B_OK. Success.
- B_BAD_VALUE. Invalid *path* specified; it may be NULL or empty.
- B_ENTRY_NOT_FOUND. The specified *path* does not exist.

- B_NAME_TOO_LONG. The *path* specified is too long.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_NO_MEMORY. Insufficient memory to perform the operation.
- B_FILE_ERROR. An invalid file prevented the operation.

GetEntry()

```
status_t GetEntry(BEntry *entry) const
```

Initializes *entry* to represent this BDirectory. If the initialization fails, *entry* is Unset().

RETURN CODES

- B_OK. Success.
- B_NAME_TOO_LONG. The path specified by *entry* is too long.
- B_ENTRY_NOT_FOUND. The specified path does not exist.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_BAD_VALUE. *entry* is uninitialized.
- B_NO_MEMORY. Insufficient memory to perform the operation.
- B_BUSY. A busy node could not be accessed.
- B_FILE_ERROR. An invalid file prevented the operation.
- B_NO_MORE_FDS. All file descriptors are in use (too many open files).
- B_NOT_A_DIRECTORY. The path includes non-directory entries.

GetNextDirents() see GetNextEntry()

GetNextEntry() , GetNextRef() , GetNextDirents() , CountEntries() , Rewind()

```
virtual status_t GetNextEntry(BEntry *entry, bool traverse = false)
virtual status_t GetNextRef(entry_ref *ref)
virtual int32 GetNextDirents(dirent *buf, size_t bufsize, int32 count = INT_MAX)
virtual int32 CountEntries(void)
virtual status_t Rewind(void)
```

The three GetNext...() functions retrieve the "next" entry that lives in the BDirectory and returns it as a BEntry, entry_ref, or dirent structure.

- GetNextEntry() returns the entry as a BEntry object. If *traverse* is true and the entry is a symbolic link, the link is traversed. In other words, *entry* could end up being in a different directory than the one referred to by this. When all entries have been visited, the function returns B_ENTRY_NOT_FOUND. The *entry* argument must be allocated before it's passed in.

- GetNextRef() return the next entry in *ref*. Since an entry_ref doesn't supply enough information to determine if the entry is a link, there's no question of traversal: The entry_ref points to *exactly* the next entry. When all entries have been visited, the function returns B_ENTRY_NOT_FOUND. The *ref* argument must be allocated before it's passed in.
- GetNextDirents() returns some number of dirent structures, either as many as can be stuffed into *buf* (where *bufsize* gives the size of *buf*), or *count* structures, whichever is smaller. The function returns the number of structures that were stuffed into *buf*; when all entries have been visited, it returns 0.

Currently, GetNextDirents() only reads one dirent at a time, no matter how many you ask for.

GetNextEntry() and GetNextRef() are reasonably clear; the dirent version deserves more explanation. You'll find this explanation (and an example) in the BEntryList class. Also, keep in mind that the set of candidate entries is different for the dirent version: GetNextDirents() finds *all* entries, including the entries for "." and "..". The other two versions skip these entries.

When you're done reading the BDirectory's entries, you can rewind the object's entry iterator by calling Rewind().

CountEntries() returns the number of entries (not counting "." and "..") in the directory.

Never call CountEntries() while you're iterating through the directory. CountEntries() does a rewind, iterates through the entries, and then rewinds again.

RETURN CODES

- B_OK. Success.
- B_FILE_ERROR. BDirectory object has not been properly initialized.
- B_NOT_A_DIRECTORY. The directory is invalid.
- B_NAME_TOO_LONG. The dirent's name is too long.
- B_ENTRY_NOT_FOUND. End of directory reached.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_BAD_VALUE. Invalid input specified, or BDirectory object has not been properly initialized.
- B_NO_MEMORY. Insufficient memory to perform the operation.

GetNextRef() see GetNextEntry()

GetStatFor()

```
status_t GetStatFor(const char *path, stat *st) const
```

Gets the stat structure for the entry designated by *path*. *path* may be either absolute or relative; if it's relative, it's reckoned off of the BDirectory's directory. This is, primarily, a convenience function; but it's also provided for efficiency.

RETURN CODES

- B_OK. Success.
- B_FILE_ERROR. An invalid file prevented the operation.
- B_NAME_TOO_LONG. The *path* specified is too long.
- B_ENTRY_NOT_FOUND. The specified *path* does not exist.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_BAD_VALUE. Invalid input specified; the *path* may be NULL or empty.
- B_NO_MEMORY. Insufficient memory to perform the operation.

IsRootDirectory()

```
bool IsRootDirectory(void) const
```

Returns true if this BDirectory represents a *root directory*. A root directory is the directory that's at the root of a volume's file hierarchy. Every volume has exactly one root directory; all other files in the volume's hierarchy descend from the root directory.

Rewind() see GetNextEntry()

SetTo() , Unset()

```
status_t SetTo(const entry_ref *ref)
status_t SetTo(const node_ref *nref)
status_t SetTo(const BEntry *entry)
status_t SetTo(const char *path)
status_t SetTo(const BDirectory *dir, const char *path)
void Unset(void)
```

Closes the BDirectory's current directory (if any), and initializes the object to open the directory as given by the arguments.

- In the *path* version, *path* can be absolute or relative, and can contain "." and ".." elements. If *path* is relative, it's reckoned off of the current working directory.
- In the *dir*/*path* version, *path* must be relative. It's reckoned off of the directory given by *dir*.

If the specification results in a symbolic link that resolves to a directory, then the linked-to directory is opened. If the specification is (or resolves to) a regular file, the initialization fails.

RETURN CODES

- B_OK. Success.
- B_NAME_TOO_LONG. The *path* specified is too long.
- B_ENTRY_NOT_FOUND. The directory does not exist.
- B_LINK_LIMIT. A cyclic loop has been detected in the file system.
- B_BAD_VALUE. Invalid input specified.
- B_NO_MEMORY. Insufficient memory to perform the operation.

- **B_BUSY. A busy node could not be accessed.**
- **B_FILE_ERROR. An invalid file prevented the operation.**
- **B_NO_MORE_FDS. All file descriptors are in use (too many open files).**

Operators

= (assignment)

```
BDirectory& operator=(const BDirectory &directory)
```

In the expression

```
BDirectory a = b;
```

BDirectory *a* is initialized to refer to the same directory as *b*. To gauge the success of the assignment, you should call InitCheck() immediately afterwards. Assigning a BDirectory to itself is safe.

Assigning from an uninitialized BDirectory is "successful": The assigned-to BDirectory will also be uninitialized (B_NO_INIT).

C Functions

create_directory()

```
status_t create_directory(const char *path, mode_t mode)
```

Creates all missing directories along the path specified by *path*.

- The pathname can be absolute or relative. If it's relative, the path is reckoned of the current working directory. If any symlinks are found in the existing portion of the path, they're traversed.
- *path* can contain ".", but it may not contain "..".
- *mode* is the permissions setting (typically expressed as an octal number) that's assigned to all directories that are created. To set the directories to be readable, writable, and "enterable" by all (for example), you would set the mode to 0777.

RETURN CODES

- B_OK. *path* now fully exists (or did in the first place).
- B_BAD_VALUE. *path* is NULL, is empty, or contains "..".
- B_NOT_ALLOWED. Read-only volume.
- B_NO_MEMORY. Insufficient memory to perform the operation.

---

find_directory()

Declared in: be/storage/FindDirectory.h

```
status_t find_directory(directory_which which, dev_t volume, bool create_it,
      char *path_string, int32 length)
status_t find_directory(directory_which which, BPath *path_obj,
      bool create_it = false, BVolume *volume = NULL)
```

The first version of this function can be used in either C or C++ code. The second version is for C++ code only.

Finds the path to the directory symbolized by *which* and copies it into *path_string*, or uses it to initialize *path_obj*.

- The *create_it* argument tells the function to create the directory if it doesn't already exist.
- *volume* identifies the volume (as a dev_t identifier or BVolume object) on which you want to look. The C++ default (NULL) means to look in the boot volume.
- The *length* argument (first version only) gives the length of *path*.

The directory_which constants are described in the "Global Constants and Defined Types" section at the end of this chapter.

RETURN CODES

- B_OK. The directory was found.

- ***Other codes.*** **The directory wasn't found or couldn't be created.**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDirectory Function Summary

Function Summary

Function	Description
=	Causes one directory to refer to the same directory as the other
Contains()	Tests whether a file is contained within the directory
CountEntries()	Returns the number of entries in the directory
CreateDirectory()	Creates a new directory
CreateFile()	Creates a new file
CreateSymLink()	Creates a new symbolic link
create_directory()	Creates all directories along a given path
BDirectory()	Constructor
~BDirectory()	Destructor
FindEntry()	Returns the entry_ref for a file with a given name
find_directory()	Finds the path to a given directory
GetEntry()	Returns a BEntry for the directory
GetNextDirents()	Fetches the dirent's for the next files in the directory
GetNextEntry()	Fetches the BEntry for the next file in the directory
GetNextRef()	Fetches the entry_ref for the next file in the directory
GetStatFor()	Returns the stat structure for a file

IsRootDirectory()	Tests whether the directory is a root directory
Rewind()	Resets the directory iterator
SetTo()	Initializes the object to point to a directory
Unset()	Uninitializes the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BEntryList

Derived from: *none*
Declared in: be/storage/EntryList.h
Library: *none*
Summary: *more...*

BEntryList is a pure abstract class that defines the protocol for iterating over a set of file system entries. Each derived class must figure out how to create (or "discover") the entry list in the first place: BEntryList only supplies functions for getting entries out of the list, it doesn't let you put them in. The BEntryList class has two derived classes: BDirectory and BQuery.

At the heart of the BEntryList class are the three **`GetNext...()`** functions, which let you retrieve the entries as...

- BEntry objects (**`GetNextEntry()`**),
- **`entry_ref`** structures (**`GetNextRef()`**),
- or **`dirent`** ("directory entry") structures (**`GetNextDirents()`**).

You call these functions iteratively; each call gets the "next" entry (or set of entries in the case of **`GetNextDirents()`**). You check the `GetNext...()` return value to detect the end of the list:

- For **`GetNextEntry()`** and **`GetNextRef()`**, **`B_ENTRY_NOT_FOUND`** indicates that there are no more entries to get.
- **`GetNextDirents()`** returns 0 when it's at the end of the list.

To get back to the top of an entry list, you call **`Rewind()`**, but note...

`Rewind()` applies to BDirectories only. You can't rewind a BQuery's entry list.

Here's an example of an iteration over all the entries in a BDirectory, retrieved as BEntry objects:

```
BDirectory dir("/boot/home/fido");
BEntry entry;

dir.Rewind();
while (dir.GetNextEntry(&entry) == B_OK)
   /* do something with entry here. */
```

The final BEntryList function, **`CountEntries()`**, also only applies to BDirectories; but even there you shouldn't depend on it. The count is stale as soon as **`CountEntries()`** returns. The user could create a new file or delete a file in the directory while you're iterating over the entries.

Also, **CountEntries()** shares the entry list pointer with the **GetNext...()** functions. You mustn't intermingle a **CountEntries()** call within your **GetNext...()** loop.

One more BDirectory wrinkle:

- Entries are retrieved in "directory order". (This is a POSIX term that means, roughly, ASCII order.) If the user renames a file while you're iterating over the directory, it's possible that the file won't be seen, or will show up under its old name *and* its new name.

The Entry List Pointer

Each BEntryList object has a single iterator pointer that's shared by all three **GetNext...()** formats (and **CountEntries()**). Thus, each successive call to a `GetNext...()` function gets the next entry, regardless of the format. For example:

```
BEntry entry;
entry_ref ref;

dir.GetNextEntry(&entry);
dir.GetNextRef(&ref);
```

Here, `entry` represents the first entry in the directory, and `ref` represents the second entry.

Multiple Retrieval

GetNextDirents() is different from the other two flavors in that it can retrieve more than one entry at a time. Or it will, someday; currently **GetNextDirents()** retrieves only one entry at a time, no matter how many you ask for.

Choosing an Iterator

So, which flavor of `GetNext...()` should you use? Here's how they compare:

- **GetNextDirents()** is by far the fastest (even in the current one-struct-at-a-time version), but it's also the least wieldy—the protocol isn't nearly as nice as the other two functions. The **dirent** structure, while jam-packed with fun facts, usually has to be turned into other structures (**node_ref**s or **entry_ref**s) in order to be useful.
- **GetNextRef()** is slower, but the **entry_ref** structure can be immediately usable (or, at least, cachable). Nonetheless, you're still a step away from a "real" object.
- **GetNextEntry()** is the slowest, but at least it hands you an object that you can sink your teeth into.

The actual timing numbers depend on your machine, the class that you're invoking the functions through, and some other factors. But the difference is (ahem) significant: **GetNextDirents()** is about an order of magnitude faster than **GetNextEntry()**, with **GetNextRef()** right about in the middle.

If, for example, you're simply compiling a list of leaf names, you should certainly use **GetNextDirents()** (painful though it may be). But if you plan on actually *doing* something with each and every entry that you retrieve, then bite the bullet: Use **GetNextEntry()**.

The dirent Structure and GetNextDirents()

Of the three iterator functions, **GetNextDirents()** needs some explanation. The **dirent** structure, which is what the function returns, describes aspects of the retrieved entry:

```
typedef struct dirent {
   dev_t d_dev;
   ino_t d_ino;
   dev_t d_pdev;
   ino_t d_pino;
   unsigned short d_reclen;
   char d_name[1];
} dirent;
```

The fields are:

- `d_dev` is a device id that identifies the device (file system) on which this entry lies.
- `d_ino` is the node number for this entry's node.
- `d_pdev` and `d_pino` are the device and inode numbers for the parent directory.
- **`d_reclen`** is the length of this dirent structure. The length is variable because...
- **`d_name`** is a buffer that's allocated to hold the (**`NULL`**-terminated) name of this entry.

So—let's pretend we've retrieved a dirent and we want to do something with it. In addition to looking at individual fields, we can combine some of them to make other structures:

- `d_dev` + `d_ino` = **node_ref** of the entry's node
- `d_pdev` + `d_pino` = **node_ref** of the parent directory
- `d_pdev` + `d_pino` + `d_name` = **entry_ref** for the entry

In code:

```
dirent *dent;
entry_ref ref;
node_ref nref;
node_ref pnref;

/* Allocate and fill the dirent here... */
...

/* Make a node_ref to this entry's node. */
```

```
nref.device = dirent->d_dev;
nref.node = dirent->d_ino;

/* Make a node_ref to this entry's parent. */
pnref.device = dirent->d_pdev;
pnref.node = dirent->d_pino;

/* Make an entry_ref to this entry. */
ref.device = dirent->d_pdev;
ref.directory = dirent->d_pino;
ref.set_name(dirent->d_name);
```

Where you go from here is a simple matter of programming. Me? I'm going to lunch.

Getting a dirent

Now that we know what to do with a **dirent**, let's see how to get one. The **GetNextDirents()** protocol looks like this:

- int32 `GetNextDirents(`dirent *_buf_, size_t _bufsize_, int32 _count_ = _INT_MAX_`)`

By default, the function stuffs as many **dirent** structs as it can into the first *bufsize* bytes of *buf*. These structures represent the next *N* entries in the entry list. The *count* argument lets you set a limit to the number of structures that you want to be retrieved at a time. The function returns the number of structures that it actually got.

Keep in mind that currently **GetNextDirents()** can only read one **dirent** at a time, regardless of the size of *buf*, or the value of *count*.

Let's try it. For the purposes of this example, we'll convert each **dirent** into an **entry_ref**, as described in the previous section.

```
/* This is the buffer that we'll stuff structures into. */
char buf[4096];
dirent *dent;
entry_ref ref;

/* We'll assume dir is a valid BDirectory object. */
while ((count = dir.GetNextDirents((dirent *)buf, 4096)) > 0) {
   dent = (dirent *)buf;

   /* Now we step through the dirents. */
   while (count-- > 0) {
      ref.device = dent->d_pdev;
      ref.directory = dent->d_pino;
      ref.set_name(dent->d_name);
```

```
      /* Do something with the ref. */
      ...

      /* Bump the pointer. */
      dent = (dirent *)((char *)dent + dent->d_reclen);
   }
}
```

Remember, the structure is variable length—you have to increment the pointer by hand, as shown here.

Member Functions

CountEntries()

```
virtual int32 CountEntries(void) = 0
```

Returns the number of entries that are in the entry list.

For BQuery this is a no-op. Also, BDirectory's implementation manipulates the entry list pointer; thus, you shouldn't call `CountEntries()` while you're iterating through the directory's entries.

GetNextEntry() , GetNextRef() , GetNextDirents()

```
virtual status_t GetNextEntry(BEntry *entry, bool traverse = false) = 0
virtual status_t GetNextRef(entry_ref *ref) = 0
virtual int32 GetNextDirents(dirent *buf, size_t bufsize, int32 count = INT_MAX) = 0
```

These functions return the "next" entry in the entry list as a BEntry, `entry_ref`, or **`dirent`** structure. The end of the list is signalled by...

- `GetNextEntry()` and `GetNextRef()` return **`B_ENTRY_NOT_FOUND`**.
- `GetNextDirents()` returns 0.

If the *traverse* argument is **`true`** , symlinks are traversed.

See the introduction for more information.

Rewind()

```
virtual status_t Rewind(void) = 0
```

Rewinds the entry list pointer so it points to the first element in the list.

For BQuery this is a no-op.

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BEntryList Function Summary

Function Summary

Function	Description
CountEntries()	Returns the number of entries in the entry list
GetNextDirents()	Returns the next entries in the entry list as dirents
GetNextEntry()	Returns the next entry in the entry list as a BEntry
GetNextRef()	Returns the next entry in the entry list as an entry_ref
Rewind()	Resets the entry list iterator

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BQuery

Derived from: BEntryList
Declared in: be/storage/Query.h
Library: libbe.so
Summary: *more...*

A *query* is a means of asking the file system for a set of entries that satisfy certain criteria. As examples, you can ask for all the entries with names that start with a certain letter, or that have nodes that are bigger than a certain size, or that were modified within the last *N* days, and so on.

The BQuery class lets you create objects that represent specific queries. To use a BQuery you have to follow these steps:

1. **Initialize**. The first thing you have to do is initialize the object; there are two parts to the initialization: You have to set the volume that you want to query over (`SetVolume()`), and set the query's "criteria formula" (`SetPredicate()`)

2. **Fetch**. After the BQuery has been properly initialized, you invoke `Fetch()`. The function returns immediately while the query executes in the background.

3. **Read**. As soon as `Fetch()` returns, you can start reading the list of winning entries by making iterative calls to the entry-list functions `GetNextRef()`, `GetNextEntry()`, and `GetNextDirents()`. If you ask for entries faster than the query can deliver them, your `GetNext...()` call will block until the next entry arrives. The function returns an error when there are no more entries to retrieve.

The set of entries that the `GetNext...()` calls retrieve (for a given fetch) are called the query's "static" entries. This distinction will become useful when we speak of "live" queries, below.

Reusing your BQuery

Want to go around again? You can, but first you have to clear the object:

- Between each "fetching session," you have to invoke `Clear()` on your BQuery object.

Clearing erases the object's predicate, volume, target (which we'll get to later), and list of static entries—in other words, clearing gets you back to a fresh BQuery object.

And speaking of going around again, be aware that the `Rewind()` function, which BQuery inherits from BEntryList, is implemented to be a no-op: **You can't rewind a BQuery's list of static entries.** After you've performed a fetch, you should read the entry list as quickly as possible and get on with things; you can't turn back or start over.

`CountEntries()` **is also a no-op**. This function is also defined by BEntryList. It doesn't apply to BQueries.

Live Queries

A live query is the gift that keeps on giving. After you tell a live query to fetch, you walk through the entry list (as described above), and then you wait for "query update" messages to be sent to your "target." A query update message describes a single entry that has changed so that...

- it now satisfies the predicate (where it didn't use to), or...
- it no longer satisfies the predicate (where it did before).

Not every BQuery is live; you have to tell it you want it to be live. To do this, all you have to do is set the object's target, through the **`SetTarget()`** function. The target is a BMessenger that identifies a BHandler/BLooper pair (as described in the **`SetTarget()`** function). Also...

- **Live query notifications stop when you `Clear()` or destroy the BQuery object.**

Another important point regarding live queries is that you can start receiving updates *before* you're done looking at all the static entries (in other words, before you've reached the end of the **`GetNext...()`** loop). It's possible that your target could receive an "entry dropped out" update before you retrieve the entry through a `GetNext...()` call. If you're using live queries, you should take care in synchronizing the `GetNext...()` iteration with the target's message processing.

We'll look at the format of the update message in a moment; first, let's fill in some gaps.

The Predicate, Attributes, and Indices

A BQuery's predicate is a logical expression that evaluates to true or false. The "atoms" of the expression are comparisons in the form...

- *attribute op value*

...where *attribute* is the name of an existing attribute, *op* is a constant that represents a comparison operation (==, <, >, etc), and *value* is the value that you want to compare the attribute to.

Attributes

As mentioned above, the attribute part of a query is a string name. When you tell the query to fetch, the file system looks for all nodes that have an attribute with that name and then compare the attribute's value to the appropriate value in the predicate. However...

- **Every query must include at least one indexed attribute.**
- **The index only knows about attributes that were written *after the index (for that attribute) was created***.

To index an attribute, you call the **`fs_create_index()`** function. Unfortunately, there's currently no way to retroactively include existing attributes in a newly created index. (Such a utility

would be simple enough to write, but it would take a long time to execute since it would have to look at every file in the file system.)

Only string and numeric attributes can be queried. Although an attribute can hold any type of data (it's stored as raw bytes), the query mechanism can only perform string and numeric comparisons.

On the bright side, every file gets three attributes for free:

- "name" is the name of the entry.
- "size" is the size of the data portion of the entry's node. The size is a 64-bit integer, and doesn't include the node's attributes.
- "last_modified" is the time the entry's node was last modified (data and attributes), measured in seconds since January 1, 1970. The modification time is recorded as a 32-bit integer.

Technically, "name", "size", and "last_modified" aren't actually attributes—you can't get them through `BNode::ReadAttr()`, for example. But they're always eligible as the attribute component in a query.

Values

The *value* part of the "*attribute op value*" equation is any expression that can be evaluated at the time the predicate is set. Once evaluated, the value doesn't change. For example, you can't specify another attribute as the value component in hopes of comparing, file by file, the value of one attribute to the value of another. The value is just data. And data is data.

The type of the value should match the type of the attribute: You compare string attributes to strings; numeric attributes to numbers. You aren't prevented from comparing a string to a number (for example), but it may not give you the result you expect.

The value of an indexed attribute can be, at most, 255 bytes.

Constructing a Predicate

There are two ways to construct a predicate:

- You can set the predicate formula as a string through SetPredicate(), or
- You can construct the predicate by "pushing" the components in Reverse Polish Notation (or "postfix") order through the `PushAttr()`, `PushValue()`, and `PushOp()` functions. There are seven value-pushing functions that push specific types: string, int32, uint32, int64, uint64, float, and double.

You can't combine the methods: Pushing the predicate *always* takes precedence over `SetPredicate()`, regardless of the order in which the methods are deployed.

SetPredicate() features:

- Comparison operators: = < > <= >= !=
- Logical operators: || &&
- Negation operator: !
- Grouping: ()
- String (value) wildcard: * (prefix and/or postfix only)
- String (value) quoting: ' '

The following are all legitimate strings that you can pass to `SetPredicate()`:

- size < 500

  (name = fido) || (size >= 500)

  (! ((name = *id*) || ('final utterance' = 'pass the salt'))) && (last_modified > 1024563)

Push features:

- The `PushOp()` function takes operator symbols, such as `B_EQ` (equals), `B_GT` (greater than), `B_LT` (less than), and so on. The complete list is given in the `PushOp()` function description.
- Value strings passed as arguments to `PushString()` are naturally quoted, so you don't have to single-quote to embed spaces or other odd characters.
- The '*' wildcard is allowed, or you can use special "contains", "begins with", and "ends with" operators.

In Reverse Polish Notation, the operator is postfixed. You then push the components from left to right. For example, this...

- size < 500

...becomes...

size 500 <

The push sequence is...

```
query.PushAttr("size");
query.PushInt32(500);
query.PushOp(B_LT);
```

Another example; this...

- (name = fido) || (size >= 500)

...becomes...

- (name fido =) (size 500 >=) ||

In code:

```
query.PushAttr("name");
query.PushString("fido");
query.PushOp(B_EQ);
query.PushAttr("size");
query.PushInt32(500);
query.PushOp(B_GE);
query.PushOp(B_OR);
```

There are no grouping operators in this notation; they're not needed—grouping is implied by the order in which the components are pushed.

When you're performing a numeric comparison, the `Push...()` function that you choose doesn't have to exactly match the natural type of the attribute, but you can't mix integers and floating point. For example, even though "size" is a 64 bit value, you can compare it to an int32...

```
query.PushAttr("size");
query.PushInt32(2000);
query.PushOp(B_GE);
```

But you can't (or shouldn't) compare it to a float...

```
query.PushAttr("size");
query.PushInt32(2000);
query.PushOp(B_GE);
```

Query Update Messages

The BMessages that are delivered by a live query have a `what` field of **`B_QUERY_UPDATE`**. The rest of the message depends on what happened:

- If the update is telling you that an entry has passed the predicate, the message's "opcode" field will be **`B_ENTRY_CREATED`**.
- If the update is telling you that an entry has been eliminated from the query, the "opcode" field will be **`B_ENTRY_REMOVED`**.

Note that the format of the messages that a live query generates are the same as the similarly-opcode'd Node Monitor messages. The only difference is the `what` field (the what for Node Monitor messages is **`B_NODE_MONITOR`**).

Entry Created

The **`B_ENTRY_CREATED`** opcode means an entry has changed so that it now passes the query's predicate. The message's fields are:

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ENTRY_CREATED
"name"	B_STRING_TYPE	The name of the new entry.
"directory"	B_INT64_TYPE	The **ino_t** (node) number for the directory in which the entry was created.
"device"	B_INT32_TYPE	The **dev_t** number of the device on which the new entry resides.
"node"	B_INT64_TYPE	The **ino_t** number of the new entry itself. (More accurately, it identifies the node that corresponds to the entry.)

If you want to cache a reference to the entry, notice that you can create an entry_ref and a node_ref with the data in the message's fields:

```
/* Create an entry_ref */
entry_ref ref;
const char *name;
...
msg->FindInt32("device", &ref.device);
msg->FindInt64("directory", &ref.directory);
msg->FindString("name", &name);
ref.set_name(name);

/* Create a node_ref */
node_ref nref;
status_t err;

...
msg->FindInt32("device", &nref.device);
msg->FindInt64("node", &nref.node);
```

The node_ref is handy because you may want to start monitoring the node (through a call to the Node Monitor). We'll get back to this point when discussing B_ENTRY_REMOVED messages.

Entry Removed

The B_ENTRY_REMOVED opcode means an entry used to pass the predicate, but something has changed (in the entry or the entry's node) so that now it doesn't.

Field	Type code	Description
"opcode"	B_INT32_TYPE	B_ENTRY_REMOVED
"directory"	B_INT64_TYPE	The **ino_t** (node) number of the directory from which the entry was removed.
"device"	B_INT32_TYPE	The **dev_t** number of the device that the removed node used to live on.
"node"	B_INT64_TYPE	The **ino_t** number of the node that was removed.

Notice that the B_ENTRY_REMOVED message doesn't tell you the name of the entry. This is an unfortunate oversight that will be corrected. In the meantime, if you need to match the node in this message to an entry from a previous `B_ENTRY_CREATED` (or that you got from a `GetNext...()` invocation), you have to keep track of the entry/node yourself. However, the location of the entry that "contains" the node may have changed since the time that the entry passed the predicate. Follow this outline:

1. You set up a live query ask for entries that have nodes larger than 500 bytes.

2. The query mechanism tells you (either in the static set or through a `B_ENTRY_CREATED` message) that "/boot/home/fido/data" satisfies the predicate.

3. You create an `entry_ref` and a `node_ref` to the entry, and cache them away somewhere.

4. *The user then renames or moves the entry.* The query mechanism doesn't tell you about this change—it only cares about the size of the node, not its name

5. You get a `B_ENTRY_REMOVED` message. You create a `node_ref` from the message and match it to your cache—and get an out-of-date `entry_ref`.

To get around the lack of a "name" field, you should monitor the nodes that you receive in your initial `GetNext...()` calls and `B_ENTRY_CREATED` messages.

Constructor and Destructor

BQuery()

```
BQuery(void)
```

Creates a new BQuery object. To use the object, you have to set its predicate and volume, and then tell it to `Fetch()`. If you want to fetch again, you have to call `Clear()` first (and re-set the predicate and volume.)

~BQuery()

```
virtual ~BQuery()
```

Destroys the BQuery. If the query is live, the query is shot dead. You stop receiving live query updates when you delete the BQuery object.

Member Functions

Clear()

```
status_t Clear(void)
```

Erases the BQuery's predicate, sets the volume and target to **NULL**, and turns off live query updates (if the query is live). You call `Clear()` if you want to `Fetch()` more than once: You have to `Clear()` before each `Fetch()` (except the first).

RETURN CODES

`Clear()` always return `B_OK`.

CountEntries() , Rewind()

Don't use these functions. They're no-ops for the BQuery class.

Fetch()

```
status_t Fetch(void)
```

Tells the BQuery to go fetch the entries that satisfy the predicate. After you've fetched, you can retrieve the set of "static" entries through calls to `GetNextEntry()`, `GetNextRef()`, or `GetNextDirents()`.

If you've set the BQuery's target, then this query is live. The live query update messages start rolling in when you tell the object to `Fetch()`. They stop when you `Clear()` or destroy the object.

The fetch fails if the object's predicate or volume isn't set, or if you've already fetched but haven't `Clear()`'d since then.

RETURN CODES

- `B_OK`. The fetch is running.
- `B_NO_INIT`. The volume or predicate isn't set.
- `B_BAD_VALUE`. The predicate is improper.
- `B_NOT_ALLOWED`. You've already fetched; `Clear()` the object and start again.

Every query must include at least one indexed attribute. If your predicate includes no indexed attributes, `Fetch()` will not balk—it returns `B_OK` (given that it doesn't otherwise fail). However, no

entries will have been retrieved, and your subsequent **GetNext...()** call will fail (**B_BAD_VALUE**).

GetNextEntry() , GetNextRef() , GetNextDirents()

virtual status_t **GetNextEntry(**BEntry **entry*, bool *traverse = false***)**
virtual status_t **GetNextRef(**entry_ref **ref***)**
virtual int32 **GetNextDirents(**dirent **buf*, size_t *bufsize*, int32 *count = INT_MAX***)**

These functions return the next entry in the "static" entry list; the list is created when you tell your (well-formed) BQuery to **Fetch()**. You can retrieve the entry as a BEntry, **entry_ref**, or **dirent** structure. The static entry list is the set of entries that initially satisfy the predicate; entries found by the live query mechanism are not included in this list.

When you reach the end of the entry list, the `Get...()` function returns an indicative value:

- `GetNextRef()` and `GetNextEntry()` return **B_ENTRY_NOT_FOUND**.
- `GetNextDirents()` returns 0.

You can only cycle over the list once; the **Rewind()** function is not defined for BQuery. See the BEntryList class for more information on these functions.

RETURN CODES

`GetNextDirents()` returns the number of **dirent**s it retrieved (currently, it can only retrieve one at a time. The other two functions return these codes:

- **B_OK**. The entry was retrieved.
- **B_ENTRY_NOT_FOUND**. You're at the end of the list.
- **B_BAD_VALUE**. The predicate includes unindexed attributes.
- **B_FILE_ERROR**. The BQuery hasn't fetched.

GetPredicate() see SetPredicate()

IsLive() see SetTarget()

PredicateLength() see SetPredicate()

PushAttr() , PushOp() , PushUInt32() , PushInt32() , PushUInt64() , PushInt64() , PushFloat() , PushDouble() , PushString() , query_op

void **PushAttr(**const char **attr_name***)**
void **PushOp(**query_op *operator***)**
void **PushUInt32(**uint32 *value***)**

void **PushInt32(**int32 *value***)**
void **PushUInt64(**uint64 *value***)**
void **PushInt64(**int64 *value***)**
void **PushFloat(**float *value***)**
void **PushDouble(**double *value***)**
void **PushString(**const char **string*, bool *case_insensitive = false***)**

You use these functions to construct the BQuery's predicate. They create a predicate expression by pushing attribute names, operators, and values in Reverse Polish Notation (post-fix) order.

- `PushAttr()` pushes an attribute name.
- `PushOp()` pushes one of the `query_op` operators listed below.
- The rest of the functions push values of the designated types.

For details on how the push method works, see "Constructing a Predicate."

The predicate that you construct through these functions can be returned as a string through the **`GetPredicate()`** function.

The `query_op` constants are:

Constant	Operation
B_EQ	=
B_NE	!=
B_GT	>
B_LT	<
B_GE	>=
B_LE	<=
B_CONTAINS	string contains value ("**value**")
B_BEGINS_WITH	string begins with value ("*value***")
B_ENDS_WITH	string ends with value ("***value**")
B_AND	&&
B_OR	||
B_NOT	!

Rewind() see CountEntries()

SetTarget() , IsLive()

status_t **SetTarget(**BMessenger *target***)**

bool **IsLive(**void**)** const

Sets the BQuery's target. The target identifies the BLooper/BHandler pair (*a la* the BInvoker target protocol) that will receive subsequent live query update messages. Calling this function declares the query to be live.

If *target* is **NULL**, the BQuery is told to be "not live". However, you can only turn off liveness (in this way) before you **Fetch()**. In other words, if you set the target, and then call **Fetch()** and *then* call `SetTarget(NULL)`, the BQuery will think that it (itself) is not live, but it really is.

IsLive() tells you if the BQuery is live. The "liveness" needn't be actuated yet—live queries don't start operating until you tell the BQuery to **Fetch()**. The live query is killed when you delete or **Clear()** the BQuery object.

RETURN CODES

- **B_OK**. The target was set (including set to **NULL**).
- **B_BAD_VALUE**. target doesn't identify a proper looper/handler pair.
- **B_NOT_ALLOWED**. You've already **Fetch()**'d; you need to **Clear()**.

SetPredicate() , GetPredicate() , PredicateLength()

status_t **SetPredicate(**const char *_expr_**)**

status_t **GetPredicate(**char *_buf_, size_t _length_**)**

size_t **PredicateLength(**void**)**

SetPredicate() sets the BQuery's predicate as a string. Predicate strings can be simple, single comparison expressions:

```
"name = fido"
```

Or they can be more complex:

```
"((name = fid*) || (size > 500)) && (last_modified < 243567)"
```

For the complete rules on setting the predicate as a string, see "Constructing a Predicate."

You can also set the predicate through the `Push...()` functions. You can't combine the methods: Pushing the predicate *always* takes precedence over **SetPredicate()**, regardless of the order in which the methods are deployed.

`GetPredicate()` copies the predicate into *buf*; *length* gives the length of *buf*, in bytes. If you want to find out how much storage you need to allocate to accommodate the predicate, call `PredicateLength()` first.

If you set the predicate through the **`Push...()`** functions, `GetPredicate()` converts the pushed construction into a string, and returns a copy of the string to you.

`PredicateLength()` returns the length of the predicate string, regardless of how it's created.

GetPredicate() and PredicateLength() both clear the push stack. This is important, because it means that you can't build up a portion of your predicate, then call `GetPredicate()`, build a little more, look again, build some more, etc. When you call `GetPredicate()`, you're done. Your next step should be a `Fetch()`.

RETURN CODES

- `B_OK`. The predicate was successfully set or gotten.
- `B_NO_INIT`. (`Get`) The predicate isn't set.
- `B_BAD_VALUE`. (`Get`) *length* is shorter than the predicate's length.
- `B_NOT_ALLOWED`. (`Set`) You've already `Fetch()`'d; you have to `Clear()`.
- `B_NO_MEMORY`. (`Set`) Not enough memory to store the predicate string.

SetVolume()

status_t **SetVolume(**const BVolume ***volume*****)**

A query can only look in one volume at a time. This is where you set the volume that you want to look at.

RETURN CODES

- `B_OK`. The volume was set.
- `B_NOT_ALLOWED`. You've already fetched, you need to `Clear()` before you can reset the volume.

Currently, `SetVolume()` doesn't complain if *volume* is invalid. However, the subsequent `Fetch()` will fail (`B_NO_INIT`).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BQuery Function Summary

Function Summary

Function	Description
Clear()	Clears the query
CountEntries()	Do not use
Fetch()	Obtain matches to the query
GetNextDirents()	Returns the next entries in the list as dirents
GetNextEntry()	Returns the next entry in the list as a BEntry
GetNextRef()	Returns the next entry in the list as an entry_ref
GetPredicate()	Returns the predicate as a string
IsLive()	Tests whether the query is live
PredicateLength()	Returns the length of the predicate as a string
PushAttr()	Push an attribute name onto the predicate stack
PushDouble()	Push a double to the predicate stack
PushFloat()	Push a float to the predicate stack
PushInt32()	Push an int32 to the predicate stack
PushInt64()	Push an int64 to the predicate stack
PushOp()	Push a query operator onto the predicate stack
PushString()	Push a string to the predicate stack
PushUInt32()	Push a uint32 to the predicate stack

PushUInt64()	Push a uint64 to the predicate stack
BQuery()	Constructor
~BQuery()	Destructor
Rewind()	Do not use
SetPredicate()	Sets the predicate as a string
SetTarget()	Set the target of the notification messages
SetVolume()	Sets the volume for the query

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessenger

Derived from: *none*
Declared in: be/app/Messenger.h
Library: libbe.so
Allocation: Stack or constructor
Summary: *more...*

A BMessenger represents and sends messages to a *message target*, where the target is a BLooper and, optionally, a specific BHandler within that looper. The target can live in the same application as the BMessenger (a *local target*), or it can live in some other application (a *remote target*).

BMessenger's most significant function is **`SendMessage()`**, which sends its argument BMessage to the target.

For a local target, **`SendMessage()`** is roughly equivalent, in terms of efficiency, to posting a message directly to the BMessenger's target (i.e **`BLooper::PostMessage()`**).

The global **`be_app_messenger`** BMessenger pointer, which targets **`be_app`**'s main message loop, is automatically initialized for you when you create your BApplication object. You can use it wherever BMessengers are called for.

Constructor and Destructor

BMessenger()

```
BMessenger(const BHandler *handler,
    const BLooper *looper = NULL,
    status_t *error = NULL)
BMessenger(const char *signature,
    team_id team = -1,
    status_t *error = NULL)
BMessenger(const BMessenger &messenger)
BMessenger(void)
```

Creates a new BMessenger and sets its target to a local *looper*/*handler*, to the (running) application identified by *signature* or *team*, or to the target of some other *messenger*.

- **Looper/handler**. To target a looper, supply a *looper* and pass a **NULL** *handler*. When the messenger sends a message, the message will be handled by *looper*'s preferred handler. If you want the message to be sent to a specific handler within a looper, supply a *handler* and pass a **NULL** *looper.* The handler must already be attached to a looper, and can't switch loopers after this BMessenger is constructed.
- **Signature or team**. If you supply a *signature* but leave *team* as –1, the messenger targets an app with that signature. (The app must already be running; in the case of multiple instances of a running app, the exact instance is indeterminate) If you supply a *team* but no *signature*, you target exactly that team, regardless of signature. By supplying both a *team* and a *signature*, you can specify a specific instance of an app. In this case, *team* must be an app that has the proper *signature*.

Messages sent to a remote target are received and handled by the remote application's BApplication object.

The BMessenger doesn't own its target.

RETURN CODES

The constructor places an error code in *error* (if provided).

- **B_OK**. The target was properly set.
- **B_BAD_VALUE**. The application identified by *signature* couldn't be found, or both *handler* and *looper* are invalid.
- **B_BAD_TEAM_ID**. Invalid *team*.
- **B_MISMATCHED_VALUES**. *team* isn't a *signature* app, or *handler* is associated with a BLooper other than *looper*.

- **B_BAD_HANDLER**. *handler* isn't associated with a BLooper (

~BMessenger()

```
~BMessenger()
```

Frees the BMessenger; the target isn't affected.

Member Functions

IsTargetLocal() see Target()

IsValid()

```
bool IsValid(void) const
```

Returns **true** if the target looper, whether local or remote, still exists.

This function doesn't tell you whether the looper is actually ready to receive messages, or whether the handler (if it was specified in the constructor) exists. In other words, a valid BMessenger is no guarantee that a message will actually get to the target.

LockTarget() , LockTargetWithTimeout()

```
bool LockTarget(void) const
status_t LockTargetWithTimeout(bigtime_t timeout) const
```

These functions apply to local targets only.

These functions attempt to lock the target looper in the manner of the similarly named BLooper functions (see **BLooper::LockTarget()**). In addition to the error codes reported there, these functions return **false** and **B_BAD_VALUE** (respectively) if the target isn't local, or if the looper is otherwise invalid.

SendMessage()

```
status_t SendMessage(BMessage *message,
    BMessage *reply,
    bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT,
    bigtime_t replyTimeout = B_INFINITE_TIMEOUT) const
status_t SendMessage(BMessage *message,
    BHandler *replyHandler = NULL,
    bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT) const
status_t SendMessage(BMessage *message,
    BMessenger *replyMessenger,
    bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT) const
status_t SendMessage(uint32 command, BMessage *reply) const
status_t SendMessage(uint32 command, BHandler *replyHandler = NULL) const
```

Sends a copy of *message* (or a BMessage based on a *command* constant) to the object's target. The caller retains ownership of *message*. The function doesn't return until the message has been delivered; if you're sending a *message* (as opposed to a *command* constant) you can set a microsecond delivery timeout through *deliveryTimeout*.

The target can respond to the message:

- If you supply a *reply* BMessage, the response is synchronous, with an optional timeout (*replyTimeout*) that starts ticking after the original message has been delivered. If the response times out, or the target deletes the original message without responding, the *reply*->**what** is set to **B_NO_REPLY**. The caller is responsible for allocating and freeing *reply*. *message* and *reply* can be the same object.

Use caution when requesting a synchronous reply: If you call **SendMessage()** from the target looper's thread, you'll deadlock (or, at best, time out).

- If you supply a reply target (*replyMessenger* or *replyHandler*), the response is asynchronous, and is sent to the reply target.
- If you supply neither a reply message nor a reply target, the target's response is sent to **be_app_messenger**.

RETURN CODES

- B_OK. The message was delivered (and the synchronous reply was received, if applicable).
- **B_TIMED_OUT**. *deliveryTimeout* expired; the message never made it to the target.
- **B_WOULD_BLOCK**. You requested a 0 *deliveryTimeout*, and the target's message queue is full.
- **B_BAD_PORT_ID**. The messenger's target is invalid, or the reply port was deleted while waiting for a reply (synchronous response requests only).
- **B_NO_MORE_PORTS**. You asked for a synchronous reply, but there are no more reply ports.

If you specified a *handler* when you constructed your BMessenger, and if that handler has since changed loopers, **SendMessage()** won't deliver its message, but it doesn't complain (it returns **B_OK**).

Target() , IsTargetLocal() , Team()

BHandler ***Target**(BLooper ***looper**) const
bool **IsTargetLocal**(void) const
inline team_id **Team**(void) const

Target() returns the BMessenger's handler (directly) and looper (by reference in *looper*). This function only works for local targets. If **Target()** returns **NULL**, it can mean one of four things:

- The target is remote; *looper* is set to **NULL**.
- The BMessenger hasn't been initialized; *looper* is set to **NULL**.
- The handler is the looper's preferred handler; *looper* will be valid.
- The handler has been deleted; *looper* will be valid given that it hasn't been deleted as well.

`IsTargetLocal()` returns true if the target is local. `Team()` returns a target's team.

Team() see Target()

Operators

= (assignment)

```
BMessenger &operator =(const BMessenger&)
```

Sets the left-side BMessenger's target to that of the right-side object.

== (equality)

```
bool operator ==(const BMessenger&) const
```

Two BMessengers are equal if they have the same target.

The Be Book The Application Kit The Application Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessenger Function Summary

Function Summary

Function	Description
=	Provided for implementation
==	Tests whether the targets match
IsTargetLocal()	Tests whether the target is in the current team
IsValid()	Tests whether the object has a valid target
LockTarget()	Locks the target local BLooper
LockTargetWithTimeout()	Locks the target local BLooper, with timeout
BMessenger()	Constructor
~BMessenger()	Destructor
SendMessage()	Sends a BMessage to the target
Target()	Returns the target of the object
Team()	Returns the team of the target of the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessage

Derived from: *none*
Declared in: be/app/Message.h
Library: libbe.so
Allocation: new, static, or automatic
Summary: *more...*

A BMessage is a bundle of structured information. Every BMessage contains a *command constant* and some number of *data fields*.

- The command constant is an **int32** value that describes, roughly, the purpose of the BMessage. It's stored as the public **what** data member. You always set and examine the **what** value directly—you don't need to call a function. (As a convenience, you can set the command constant when you create your BMessage object.)
- The data fields are name-type-value triplets. A field is be primarily identified by name, but you can look for fields by name, type, or a combination of the two. The type is encoded as a constant (**B_INT32_TYPE**, **B_STRING_TYPE**, etc), and is meant to describe the type of value that the field holds. A single field can have only one name and one type, but can contain an array of values. Individual values in a field are accessible by index.

Neither the command constant nor the data fields are mandatory. You can create a BMessage that has data but no command, or that *only* has a command. However, creating a BMessage that has neither is pointless.

Preparatory Reading

BMessages are used throughout the kits to send data (or notifications) to another thread—possibly in another application. To understand how BMessages fit into the messaging system, see "Messaging".

The BMessage class also contributes a number of functions that help define the scripting system. See "Scripting" for an introduction to this system.

BMessages are also used by a number of classes (BClipboard, BArchivable, and others) for their ability to store data.

Types of Functions

The BMessage class defines five types of functions:

- *Data field functions*. These functions either set or retrieve the value of a data field. See **AddData(), FindData())**, **ReplaceData()**, and **RemoveName()**. .
- *Info functions*. These functions retrieve information about the state and contents of the BMessage. See **IsSystem()** and **GetInfo()**.

- *Messaging functions.* These functions are part of the messaging system. .A smaller set of functions reports on the status of a received message. For example, `IsSourceWaiting()` tells whether the message sender is waiting for a reply, `WasDropped()` says whether it was dragged and dropped, and `DropPoint()` says where it was dropped. `SendReply()`
- *Scripting functions*, such as `AddSpecifier()` and `PopSpecifier()`.
- *Flattening functions*. The data in a BMessage can be flattened. See `Flatten()`.

BMessage Ownership

The documentation for the functions that accept or pass back a BMessage object should tell you who's responsible for deleting the object. Most functions that accept a BMessage argument copy the object, leaving the caller with the responsibility for deleting the argument. The exceptions—i.e. BMessage-accepting functions that take over ownership of the object—are listed below:

Functions that return a BMessage to you usually don't give up ownership; in general, you don't delete the BMessages that are passed to you. The exceptions—functions that expect the caller to take over ownership of a passed-back BMessage—are listed below:

Data Members

uint32 **what**

A coded constant that captures what the message is about.

Constructor and Destructor

BMessage()

```
BMessage(uint32 command)
BMessage(const BMessage &message)
BMessage(void)
```

Creates a new BMessage object that has the given *command* constant, or that's a copy of another BMessage. If it's a copy, the new object contains the same command constant and data fields as *message*.

See also: `BLooper::DetachCurrentMessage()`

~BMessage()

```
virtual ~BMessage()
```

Frees all memory allocated to hold message data. If the message sender is expecting a reply but hasn't received one, a default reply (with `B_NO_REPLY` as the **what** data member) is sent before the message is destroyed.

The system retains ownership of the messages it delivers to you. Each message loop routinely deletes delivered BMessages after the application is finished responding to them.

Member Functions

AddData() , AddBool() , AddInt8() , AddInt16() , AddInt32() , AddInt64() , AddFloat() , AddDouble() , AddString() , AddPoint() , AddRect() , AddRef() , AddMessage() , AddMmessenger() , AddPointer() , AddFlat()

status_t **AddData(**const char **name*, type_code *type*, const void **data*, ssize_t *numBytes*, bool *fixedSize* = true, int32 *numItems* = 1**)**
status_t **AddBool(**const char **name*, bool *aBool***)**
status_t **AddInt8(**const char **name*, int8 *anInt8***)**
status_t **AddInt16(**const char **name*, int16 *anInt16***)**
status_t **AddInt32(**const char **name*, int32 *anInt32***)**
status_t **AddInt64(**const char **name*, int64 *anInt64***)**
status_t **AddFloat(**const char **name*, float *aFloat***)**
status_t **AddDouble(**const char **name*, double *aDouble***)**
status_t **AddString(**const char **name*, const char **string***)** status_t **AddString(**const char **name*, const BString &*string***)**
status_t **AddPoint(**const char **name*, BPoint *point***)**
status_t **AddRect(**const char **name*, BRect *rect***)**
status_t **AddRef(**const char **name*, const entry_ref **ref***)**
status_t **AddMessage(**const char **name*, const BMessage **message***)**
status_t **AddMessenger(**const char **name*, BMessenger *messenger***)**
status_t **AddPointer(**const char **name*, const void **pointer***)**
status_t **AddFlat(**const char **name*, BFlattenable **object*, int32 *numItems* = 1**)**

These functions add data to the field named *name* and assign a data type to the field. Field names can be no longer than 255 characters. If more than one item of data is added under the same name, the BMessage creates an array of data for that name. Each time you add another value (to the same name), the value is added to the end of the array—you can't add a value at a specific index. A given field can only store one type of data.

`AddData()` copies *numBytes* of *data* into the field, and assigns the data a *type* code. It copies whatever the *data* pointer points to. For example, if you want to add a string of characters to the message, *data* should be the string pointer (`char *`). If you want to add only the string pointer, not the characters themselves, *data* should be a pointer to the pointer (`char **`). The assigned *type* must be a specific data type; it should not be `B_ANY_TYPE`..

When you call **AddData()** to place the first item in an array under a new name, you can provide it with two arguments, *fixedSize* and *numItems*, that will improve the object's efficiency. If the *fixedSize* flag is `true`, each item in the array must have the same number of bytes; if the flag is `false`, items can vary in size. *numItems* tells the object to pre-allocate storage for some number of items. This isn't a limit—you can add more than *numItems* to the field.

Most of the other functions are variants of **AddData()** that hard-code the field's type. For example, **AddFloat()** assigns the type **B_FLOAT_TYPE**; **AddBool()** assigns **B_BOOL_TYPE**, and so on.

AddString(), like **AddData()**, takes a pointer to the data it adds, or you can use a BString object. The *string* must be null-terminated; the null character is counted and copied into the message. Similarly, **AddRef()** adds the pointed to **entry_ref** structure to the message (and the variable-length name that's one of the elements of the structure); **AddMessage()** adds one BMessage to another.

The other functions are simply passed the data directly. For example, **AddInt32()** takes an `int32` or `uint32` and **AddMessenger()** takes a BMessenger object, whereas **AddData()** would be passed a pointer to an `int32` and a pointer to a BMessenger. **AddPointer()** adds only the pointer it's passed, not the data it points to. To accomplish the same thing, **AddData()** would take a pointer to the pointer. (The pointer will be valid only locally; it won't be useful to a remote destination.)

AddFlat() flattens an *object* (by calling its **Flatten()** function) and adds the flat data to the message. It calls the object's **TypeCode()** function to learn the type code it should associate with the data. Objects that are added through AddFlat() must inherit from BFlattenable (defined in the Support Kit).

You can also provide a *numItems* hint to **AddFlat()** when you call it to set up a new array. **AddFlat()** calls the object's **IsFixedSize()** function to discover whether all items in the array will be the same size.

These functions return **B_ERROR** if the data is too massive to be added to the message, **B_BAD_TYPE** if the data can't be added to an existing array because it's the wrong type, **B_NO_MEMORY** if the BMessage can't get enough memory to hold the data, and **B_BAD_VALUE** if the proposed *name* for the data is longer than 255 bytes. If all goes well, they return **B_OK**.

There's no limit on the number of named fields a message can contain or on the size of a field's data. However, since the search is linear, combing through a very long list of names to find a particular piece of data may be inefficient. Also, because of the amount of data that must be moved, an extremely large message can slow the delivery mechanism. It's sometimes better to put some of the information in a common location (a file, a private clipboard, a shared area of memory) and just refer to it in the message.

See also: **FindData()**, **GetInfo()**

AddSpecifier()

status_t **AddSpecifier(**const BMessage ***message***)**
status_t **AddSpecifier(**const char ***property***)**

status_t **AddSpecifier(**const char *_property_, int32 _index_**)**
status_t **AddSpecifier(**const char *_property_, int32 _index_, int32 _range_**)**
status_t **AddSpecifier(**const char *_property_, const char *_name_**)**

Adds a specifier to the specifier stack. There are several variations of this method. The first adds the specifier *message* to the specifier stack. The other methods add a specifier targeting the property *property*, with specifier constants **B_DIRECT_SPECIFIER**, **B_INDEX_SPECIFIER**, **B_RANGE_SPECIFIER**, and **B_NAME_SPECIFIER**, respectively. For all other specifiers, you must construct the specifier separately and then call `AddSpecifier()` on the message. For more information about specifiers, see the "Scripting" section near the beginning of this chapter.

Specifiers are stored in a data array named "specifiers." However, since `AddSpecifier()` also sets the notion of the current specifier, specifiers should always be added to a scripting message with this method rather than with **AddMessage()**.

AddSpecifier() returns **B_OK** if it's able to add the specifier to the BMessage and an error code, generally only **B_NO_MEMORY** to indicate that it has run out of memory, if not.

See also: **GetCurrentSpecifier()**, **HasSpecifiers()**, **PopSpecifier()**

CountNames()

int32 **CountNames(**type_code _type_**)** const

Returns the number of named data fields in the BMessage that store data of the specified *type*. An array of information held under a single name counts as one field; each name is counted only once, no matter how many data items are stored under that name.

If *type* is **B_ANY_TYPE**, this function counts all named fields. If *type* is a specific type, it counts only fields that store data registered as that type.

See also: **GetInfo()**

DropPoint() see WasDropped()

FindData() , FindBool() , FintInt8() , FindInt16() , FindInt32() , FindInt64() , FindFloat() , FindDouble() , FindString() , FindPoint() , FindRect() , FindRef() , FindMessage() , FindMessenger() , FindPointer() , FindFlat()

status_t **FindData(**const char *_name_,
type_code _type_,
int32 _index_,
const void **_data_,
ssize_t *_numBytes_**)** const

status_t **FindData(**const char *_name_,
type_code _type_,
const void **_data_,
ssize_t *_numBytes_**)** const

status_t **FindBool(**const char *_name_,
int32 _index_,
bool *_aBool_**)** const
status_t **FindBool(**const char *_name_, bool *_aBool_**)** const

status_t **FindInt8(**const char *_name_,
int32 _index_,
int8 *_anInt8_**)** const
status_t **FindInt8(**const char *_name_,
int8 *_anInt8_**)** const

status_t **FindInt16(**const char *_name_,
int32 _index_,
int16 *_anInt16_**)** const
status_t **FindInt16(**const char *_name_, int16 *_anInt16_**)** const

status_t **FindInt32(**const char *_name_,
int32 _index_,
int32 *_anInt32_**)** const
status_t **FindInt32(**const char *_name_, int32 *_anInt32_**)** const

status_t **FindInt64(**const char *_name_,
int32 _index_,
int64 *_anInt64_**)** const
status_t **FindInt64(**const char *_name_, int64 *_anInt64_**)** const

status_t **FindFloat(**const char *_name_,
int32 _index_,
float *_aFloat_**)** const
status_t **FindFloat(**const char *_name_, float *_aFloat_**)** const

status_t **FindDouble(**const char *_name_,
int32 _index_,
double *_aDouble_**)** const
status_t **FindDouble(**const char *_name_, double *_aDouble_**)** const

status_t **FindString(**const char *_name_,
int32 _index_,
const char **_string_**)** const
status_t **FindString(**const char *_name_, const char **_string_**)** const
status_t **FindString(**const char *_name_, BString *_string_**)** const
status_t **FindString(**const char *_name_, int32 _index_, BString *_string_**)** const

status_t **FindPoint(**const char *_name_,
int32 _index_,
BPoint *_point_**)** const
status_t **FindPoint(**const char *_name_, BPoint *_point_**)** const

status_t **FindRect(**const char *_name_,
int32 _index_,

```
    BRect *rect) const
status_t FindRect(const char *name, BRect *rect) const
```

```
status_t FindRef(const char *name,
    int32 index,
    entry_ref *ref) const
status_t FindRef(const char *name, entry_ref *ref) const
```

```
status_t FindMessage(const char *name,
    int32 index,
    BMessage *message) const
status_t FindMessage(const char *name, BMessage *message) const
```

```
status_t FindMessenger(const char *name,
    int32 index,
    BMessenger *messenger) const
status_t FindMessenger(const char *name, BMessenger *messenger) const
```

```
status_t FindPointer(const char *name,
    int32 index,
    void **pointer) const
status_t FindPointer(const char *name, void **pointer) const
```

```
status_t FindFlat(const char *name,
    int32 index,
    BFlattenable *object) const
status_t FindFlat(const char *name, BFlattenable *object) const
```

These functions retrieve data from the BMessage. Each looks for data stored under the specified *name*. If more than one data item has the same name, an *index* can be provided to tell the function which item in the *name* array it should find. Indices begin at 0. If an index isn't provided, the function will find the first, or only, item in the array.

In all cases except `FindData()` and `FindString()`, the data that's retrieved from the BMessage is copied into the reference argument; the caller is responsible for freeing the copied data). For `FindData()` and the non-BString version of `FindString()`, a pointer to the data is returned; the BMessage retains ownership of the actual data and will delete the data when the object itself is deleted.

`FindData()` places, in **data*, a pointer to the requested data item. The size of the item in bytes is written to *numBytes*. If *type* is `B_ANY_TYPE`, it provides a pointer to the data no matter what type it actually is. But if *type* is a specific data type, it provides the pointer only if the *name* field holds data of that particular type.

The other functions are specialized versions of `FindData()`. They match the corresponding `Add...()` functions and search for named data of a particular type, as described below:

Function	Finds data	Registered as type
`FindBool()`	a `bool`	`B_BOOL_TYPE`
`FindInt8()`	an **int8** or **uint8**	`B_INT8_TYPE`

FindInt16()	an **int16** or **uint16**	B_INT16_TYPE
FindInt32()	an **int32** or **uint32**	B_INT32_TYPE
FindInt64()	an **int64** or **uint64**	B_INT64_TYPE
FindFloat()	a `float`	B_FLOAT_TYPE
FindDouble)	a `double`	B_DOUBLE_TYPE
FindString()	a character string	B_STRING_TYPE
FindPoint()	a BPoint object	B_POINT_TYPE
FindRect()	a BRect object	B_RECT_TYPE
FindRef()	an entry_ref	B_REF_TYPE
FindMessage()	a BMessage object	B_MESSAGE_TYPE
FindMessenger()	a BMessenger object	B_MESSENGER_TYPE
FindPointer()	a pointer to anything	B_POINTER_TYPE

The other type-specific functions retrieve the requested data item from the message by copying it to the variable referred to by the last argument; you get the data, not just a pointer to it. For example, FindMessenger() assigns the BMessenger it finds in the message to the *messenger* object, whereas FindData() would provide only a pointer to a BMessenger. FindPointer() puts the found pointer in the `void*` variable that *pointer* refers to; FindData(), as illustrated above, would provide a pointer to the pointer. (If the message was delivered from a remote source, pointers retrieved from the message won't be valid.)

FindRef() retrieves an entry_ref structure; the data that's used to reconstitute the structure may have been added as an entry_ref (through AddRef()), or as a flattened BPath object (AddFlat()).

FindFlat() assigns the object stored in the BMessage to the *object* passed as an argument—it calls the *object*'s Unflatten() function and passes it the flat data from the message—provided that the two objects have compatible types. The argument *object*'s AllowsTypeCode() function must return `true` when tested with the type code stored in the message; if not, FindFlat() fails and returns B_BAD_VALUE.

If these functions can't find any data associated with *name*, they return a B_NAME_NOT_FOUND error. If they can't find *name* data of the requested *type* (or the type the function returns), they return B_BAD_TYPE. If the *index* is out of range, they return B_BAD_INDEX. You can rely on the values they retrieve only if they return B_OK and the data was correctly recorded when it was added to the message.

When they fail, FindData() and FindString() provide `NULL` pointers. FindRect() hands you an invalid rectangle and FindMessenger() an invalid BMessenger. Most of the other functions set the data values to 0, which may be indistinguishable from valid values.

Finding a data item doesn't remove it from the BMessage.

(Several functions, such as FindRect() and FindInt32(), have versions that return the found value directly. These versions don't report errors and may not be supported in the future.)

See also: GetInfo(), AddData()

Flatten() , Unflatten() , FlattenedSize()

```
status_t Flatten(BDataIO *object, ssize_t *numBytes = NULL) const
status_t Flatten(char *address, ssize_t numBytes = NULL) const
```

```
status_t Unflatten(BDataIO *object)
status_t Unflatten(const char *address)
```

```
ssize_t FlattenedSize(void) const
```

These functions write the BMessage and the data it contains to a "flat" (untyped) buffer of bytes, and reconstruct a BMessage object from such a buffer.

If passed a BDataIO *object* (including a BFile), `Flatten()` calls the object's `Write()` function to write the message data. If passed the *address* of a buffer, it begins writing at the start of the buffer. `FlattenedSize()` returns the number of bytes you must provide in the buffer to hold the flattened object. `Flatten()` places the number of bytes actually written in the variable that its *numBytes* argument refers to.

`Unflatten()` empties the BMessage of any information it may happen to contain, then initializes the object from data read from the buffer. If passed a BDataIO *object*, it calls the object's `Read()` function to read the message data. If passed a buffer *address*, it begins reading at the start of the buffer. It's up to the caller to make sure that `Unflatten()` reads data that `Flatten()` wrote and that pointers are positioned correctly.

`Flatten()` returns any errors encountered when writing the data, or `B_OK` if there is no error.

If it doesn't recognize the data in the buffer as being a flattened object or there's a failure in reading the data, `Unflatten()` returns `B_BAD_VALUE`. If it doesn't have adequate memory to recreate the whole message, it returns `B_NO_MEMORY`. Otherwise, it returns `B_OK`.

See also: the BDataIO class in the Support Kit

GetCurrentSpecifier() , PopSpecifier()

```
status_t GetCurrentSpecifier(int32 *index,
    BMessage *specifier = NULL,
    int32 *what = NULL,
    const char **property = NULL) const
```

```
status_t PopSpecifier(void)
```

`GetCurrentSpecifier()` unpacks the current specifier in the BMessage, the one at the top of the specifier stack; `PopSpecifier()` changes the notion of which specifier is current, by popping the current one from the stack.

These functions aid in implementing a class-specific version of BHandler's `ResolveSpecifier()` function—the first gets the specifier that needs to be resolved, and the second pops it from

the stack after it is resolved. You can also call them to examine relevant specifiers when handling a message that targets an object property (such as `B_GET_PROPERTY`).

A scripting BMessage keeps specifiers in a data array named "specifiers"; each specifier is itself a BMessage, but one with a special structure and purpose in the scripting system. See the "Scripting" section near the beginning of this chapter for an overview of the system and the place of specifiers in it.

The specifiers in a message are ordered and, until `PopSpecifier()` is called, the one that was added last—the one with the greatest index—is the current specifier. `PopSpecifier()` merely decrements the index that picks the current specifier; it doesn't delete anything from the BMessage.

`GetCurrentSpecifier()` puts the index of the current specifier in the variable that its first argument, *index*, refers to. If other arguments are provided, it makes the *specifier* BMessage a copy of the current specifier. It also extracts two pieces of information from the *specifier*: It places the `what` data member of the specifier in the *what* variable and a pointer to the property name in the *property* variable. These last two output arguments won't be valid if the *specifier* argument is `NULL`.

Both functions fail if the BMessage doesn't contain specifiers. In addition, `GetCurrentSpecifier()` fails if it can't find data in the BMessage for its *specifier* and *property* arguments, and `PopSpecifier()` fails if the BMessage isn't one that has been delivered to you after being processed through a message loop. When it fails, `GetCurrentSpecifier()` returns `B_BAD_SCRIPT_SYNTAX`, but `PopSpecifier()` returns `B_BAD_VALUE`. On success, both functions return `B_OK`.

See also: `AddSpecifier()`, `HasSpecifiers()`, `BHandler::ResolveSpecifier()`

GetInfo()

```
status_t GetInfo(const char *name,
    type_code *typeFound,
    int32 *countFound = NULL) const
status_t GetInfo(const char *name,
    type_code *typeFound,
    bool *fixedSize) const
status_t GetInfo(type_code type, int32 index,
    char **nameFound,
    type_code *typeFound,
    int32 *countFound = NULL) const
```

Provides information about the data fields stored in the BMessage.

When passed a *name* that matches a name within the BMessage, `GetInfo()` places the type code for data stored under that name in the variable referred to by *typeFound* and writes the number of data items with that name into the variable referred to by *countFound*. It then

returns B_OK. If it can't find a *name* field within the BMessage, it sets the *countFound* variable to 0, and returns B_NAME_NOT_FOUND (without modifying the *typeFound* variable).

When the *fixedSize* argument is specified, the bool referenced by *fixedSize* is set to **true** if all items in the array specified by name must be the same size, and **false** if the items can be of different sizes (see AddData()).

When passed a *type* and an *index*, GetInfo() looks only at fields that store data of the requested type and provides information about the field at the requested index. Indices begin at 0 and are type specific. For example, if the requested *type* is B_DOUBLE_TYPE and the BMessage contains a total of three named fields that store `double` data, the first field would be at *index* 0, the second at 1, and the third at 2—no matter what other types of data actually separate them in the BMessage, and no matter how many data items each field contains. (Note that the index in this case ranges over fields, each with a different name, not over the data items within a particular named field.) If the requested type is B_ANY_TYPE, this function looks at all fields and gets information about the one at *index* whatever its type.

If successful in finding data of the *type* requested at *index*, GetInfo() returns B_OK and provides information about the data through the last three arguments:

- It places a pointer to the name of the data field in the variable referred to by *nameFound*.
- It puts the code for the type of data the field contains in the variable referred to by *typeFound*. This will be the same as the *type* requested, unless the requested type is B_ANY_TYPE, in which case *typeFound* will be the actual type stored under the name.
- It records the number of data items stored within the field in the variable referred to by *countFound*.

If GetInfo() can't find data of the requested *type* at *index*, it sets the *countFound* variable to 0, and returns B_BAD_TYPE. If the index is out of range, it returns B_BAD_INDEX.

This version of GetInfo() can be used to iterate through all the BMessage's data. For example:

```
char *name;
uint32 type;
int32 count;

for ( int32 i = 0;
      msg->GetInfo(B_ANY_TYPE, i, &name, &type, &count) == B_OK;
      i++ ) {
   . . .
}
```

If the index is incremented from 0 in this way, all data of the requested type will have been read when GetInfo() returns B_NAME_NOT_FOUND. If the requested type is B_ANY_TYPE, as shown above, it will reveal the name and type of every field in the BMessage.

See also: HasData(), AddData(), FindData()

HasSpecifiers()

```
bool HasSpecifiers(void) const
```

Returns `true` if the BMessage has specifiers added by an `AddSpecifier()` function, and `false` if not.

See also: `AddSpecifier()`, `GetCurrentSpecifier()`

IsEmpty() see MakeEmpty()

IsReply() see WasDelivered()

IsSourceRemote() see WasDelivered()

IsSourceWaiting() see WasDelivered()

IsSystem()

```
bool IsSystem(void) const
```

Returns `true` if the **`what`** data member of the BMessage object identifies it as a system-defined message, and `false` if not.

MakeEmpty() , IsEmpty()

```
status_t MakeEmpty(void)
bool IsEmpty(void) const
```

`MakeEmpty()` removes and frees all data that has been added to the BMessage, without altering the **`what`** constant. It returns `B_OK`, unless the message can't be altered (as it can't if it's being dragged), in which case it returns `B_ERROR`.

`IsEmpty()` returns `true` if the BMessage has no data (whether or not it was emptied by `MakeEmpty()`), and `false` if it has some.

See also: `RemoveName()`

Previous() see WasDelivered()

PrintToStream()

```
void PrintToStream(void) const
```

Prints information about the BMessage to the standard output stream (`stdout`). Each field of named data is reported in the following format,

```
#entry name, type = type, count = count
```

where *name* is the name that the data is registered under, *type* is the constant that indicates what type of data it is, and *count* is the number of data items in the named array.

RemoveName() , RemoveData()

```
status_t RemoveName(const char *name)
status_t RemoveData(const char *name, int32 index = 0)
```

`RemoveName()` removes all data entered in the BMessage under *name* and the name itself. `RemoveData()` removes the single item of data at *index* in the *name* array. If the array has just one data item, it removes the array and name just as `RemoveName()` would.

Both functions free the memory that was allocated to hold the data, and return `B_OK` when successful. However, if there's no data in the BMessage under *name*, they return a `B_NAME_NOT_FOUND` error. If message data can be read but can't be changed (as it can't for a message that's being dragged), they both return `B_ERROR`. If the *index* is out of range, `RemoveData()` returns `B_BAD_INDEX` (the index is too high) or `B_BAD_VALUE` (the value passed is a negative number).

See also: `MakeEmpty()`

ReplaceData() , ReplaceBool() , ReplaceInt8() , ReplaceInt16() , ReplaceInt32() , ReplaceInt64() , ReplaceFloat() , ReplaceDouble() , ReplaceString() , ReplacePoint() , ReplaceRect() , ReplaceRef() , ReplaceMessage() , ReplaceMessenger() , ReplacePointer() , ReplaceFlat()

```
status_t ReplaceData(const char *name,
    type_code type,
    const void *data,
    ssize_t numBytes)
status_t ReplaceData(const char *name,
    type_code type,
     int32 index,
    const void *data,
    ssize_t numBytes)
```

```
status_t ReplaceBool(const char *name, bool aBool)
status_t ReplaceBool(const char *name,
```

int32 *index*,
bool *aBool*)

status_t **ReplaceInt8**(const char **name*, int8 *anInt8*)
status_t **ReplaceInt8**(const char **name*,
int32 *index*,
int8 *anInt8*)

status_t **ReplaceInt16**(const char **name*, int16 *anInt16*)
status_t **ReplaceInt16**(const char **name*,
int32 *index*,
int16 *anInt16*)

status_t **ReplaceInt32**(const char **name*, long *anInt32*)
status_t **ReplaceInt32**(const char **name*,
int32 *index*,
int32 *anInt32*)

status_t **ReplaceInt64**(const char **name*, int64 *anInt64*)
status_t **ReplaceInt64**(const char **name*,
int32 *index*,
int64 *anInt64*)

status_t **ReplaceFloat**(const char **name*, float *aFloat*)
status_t **ReplaceFloat**(const char **name*,
int32 *index*,
float *aFloat*)

status_t **ReplaceDouble**(const char **name*, double *aDouble*)
status_t **ReplaceDouble**(const char **name*,
int32 *index*,
double *aDouble*)

status_t **ReplaceString**(const char **name*, const char **string*)
status_t **ReplaceString**(const char **name*,
int32 *index*,
const char **string*)

status_t **FindString**(const char **name*, BString &*string*)
status_t **FindString**(const char **name*, int32 *index*, BString &*string*)

status_t **ReplacePoint**(const char **name*, BPoint *point*)
status_t **ReplacePoint**(const char **name*,
int32 *index*,
BPoint *point*)

status_t **ReplaceRect**(const char **name*, BRect *rect*)
status_t **ReplaceRect**(const char **name*,
int32 *index*,
BRect *rect*)

status_t **ReplaceRef**(const char **name*, entry_ref **ref*)
status_t **ReplaceRef**(const char **name*,
int32 *index*,
entry_ref **ref*)

status_t **ReplaceMessage**(const char **name*, BMessage **message*)

```
status_t ReplaceMessage(const char *name,
    int32 index,
    BMessage *message)
```

```
status_t ReplaceMessenger(const char *name, BMessenger messenger)
status_t ReplaceMessenger(const char *name,
    int32 index,
    BMessenger messenger)
```

```
status_t ReplacePointer(const char *name, const void *pointer)
status_t ReplacePointer(const char *name,
    int32 index,
    const void *pointer)
```

```
status_t ReplaceFlat(const char *name, BFlattenable *object)
status_t ReplaceFlat(const char *name,
    int32 index,
    BFlattenable *object)
```

These functions replace a data item in the *name* field with another item passed as an argument. If an *index* is provided, they replace the item in the *name* array at that index; if an *index* isn't mentioned, they replace the first (or only) item stored under *name*. If an *index* is provided but it's out of range, the replacement fails.

`ReplaceData()` replaces an item in the *name* field with *numBytes* of *data*, but only if the *type* code that's specified for the data matches the type of data that's already stored in the field. The *type* must be specific; it can't be `B_ANY_TYPE`.

`FindFlat()` replaces a flattened object with another *object*, provided that the type reported by the argument *object* (by its `TypeCode()` function) matches the type recorded for the item in the message. If not, it returns `B_BAD_VALUE`.

The other functions are simplified versions of `ReplaceData()`. They each handle the specific type of data declared for their last arguments. They succeed if this type matches the type of data already in the *name* field, and fail if it does not. The new data is added precisely as the counterpart `Add...()` function would add it.

If successful, all these functions return `B_OK`. If unsuccessful, they return an error code—`B_ERROR` if the message is read-only (as it is while the message is being dragged), `B_BAD_INDEX` if the *index* is out of range, `B_NAME_NOT_FOUND` if the *name* field doesn't exist, or `B_BAD_TYPE` if the field doesn't contain data of the specified type.

See also: `AddData()`

ReturnAddress()

```
BMessenger ReturnAddress(void)
```

Returns a BMessenger object that can be used to reply to the BMessage. Calling the BMessenger's `SendMessage()` function is equivalent to calling `SendReply()`, except that the

return message won't be marked as a reply. If a reply isn't allowed (if the BMessage wasn't delivered), the returned BMessenger will be invalid.

If you want to use the **ReturnAddress()** BMessenger to send a synchronous reply, you must do so before the BMessage is deleted and a default reply is sent.

See also: **SendReply()**, **WasDelivered()**

SendReply()

```
status_t SendReply(BMessage *message,
    BMessage *reply,
    bigtime_t sendTimeout = B_INFINITE_TIMEOUT,
    bigtime_t replyTimeout = B_INFINITE_TIMEOUT)
status_t SendReply(BMessage *message,
    BHandler *replyHandler = NULL,
    bigtime_t sendTimeout = B_INFINITE_TIMEOUT)
status_t SendReply(uint32 command, BMessage *reply)
status_t SendReply(uint32 command, BHandler *replyHandler = NULL)
```

Sends a reply *message* back to the sender of the BMessage (in the case of a synchronous reply) or to a target BHandler (in the case of an asynchronous reply). Whether the reply is synchronous or asynchronous depends on how the BMessage that's sending the reply was itself sent:

- The reply is delivered synchronously if the message sender is waiting for one to arrive. The function that sent the BMessage doesn't return until it receives the reply (or a timeout expires). If an expected reply has not been sent by the time the BMessage object is deleted, a default **B_NO_REPLY** message is returned to the sender. If a reply is sent after the sender gave up waiting for it to arrive, the reply *message* disappears into the bowels of the system.

- The reply is delivered asynchronously if the message sender isn't waiting for a reply. In this case, the sending function designates a target BHandler and BLooper for any replies that might be sent, then returns immediately after putting the BMessage in the pipeline. Posted messages and messages that are dragged and dropped are also eligible for asynchronous replies.

SendReply() works only for BMessage objects that have been processed through a message loop and delivered to you. The caller retains ownership of the reply *message* passed to **SendReply()**; it can be deleted (or left to die on the stack) after the function returns.

SendReply() sends a message—a reply message, to be sure, but a message nonetheless. It behaves exactly like the other message-sending function, BMessenger's **SendMessage()**:

- By passing it a *reply* argument, you can ask for a synchronous reply to the reply message it sends. It won't return until it receives the reply.

- By supplying a *replyHandler* argument, you can arrange for an expected asynchronous reply. If a specific target isn't specified, the BApplication object will handle the reply if one is sent.

By default, **SendReply()** doesn't return until the reply message is delivered (placed in the BLooper's port queue). It's possible, in some circumstances, for the receiving port queue to be full, in which case **SendReply()** will block until a slot becomes free. However, you can limit how long **SendReply()** will wait to deliver the message before it gives up and returns. The *sendTimeout* argument is the number of microseconds you give the function to do its work. If the time limit is exceeded, the function fails and returns an error (**B_TIMED_OUT**).

When asking for a synchronous reply, separate *sendTimeout* and *replyTimeout* limits can be set for sending the message and receiving the reply. There is no time limit if a timeout value is set to **B_INFINITE_TIMEOUT**—as it is by default. The function won't block at all if the timeout is set to 0.

If a *command* is passed rather than a *message*, **SendReply()** constructs the reply BMessage, initializes its **what** data member with the *command* constant, and sends it just like any other reply. The *command* versions of this function have infinite timeouts; they block until the message is delivered and, if requested, a synchronous reply is received.

This function returns **B_OK** if the reply is successfully sent. If there's a problem in sending the message, it returns the same sort of error code as BMessenger's **SendMessage()**. It may also report a reply-specific problem. The more informative return values are as follows:

Error code	Is returned when
B_BAD_REPLY	Attempting to reply to a message that hasn't been delivered yet.
B_DUPLICATE_REPLY	Sending a reply after one has already been sent and delivered.
B_BAD_THREAD_ID	Sending a reply to a destination thread that no longer exists.
B_BAD_PORT_ID	Sending a reply to a BLooper and port that no longer exist.
B_TIMED_OUT	Taking longer than the specified time limit to deliver a reply message or to receive a synchronous reply to the reply.

If you want to delay sending a reply and keep the BMessage object beyond the time it's scheduled to be deleted, you may be able to detach it from the message loop. See **DetachCurrentMessage()** in the BLooper class.

See also: **BMessenger::SendMessage()**, **BLooper::DetachCurrentMessage()**, **Error**, **ReturnAddress()**

Unflatten() see Flatten()

WasDelivered() , IsSourceRemote() , IsSourceWaiting() , IsReply() , Previous()

bool **WasDelivered**(void) const
bool **IsSourceRemote**(void) const

bool **IsSourceWaiting**(void) const
bool **IsReply**(void) const
const BMessage ***Previous**(void) const

These functions can help if you're engaged in an exchange of messages or managing an ongoing communication.

WasDelivered() indicates whether it's possible to send a reply to a message. It returns `true` for a BMessage that was posted, sent, or dropped—that is, one that has been processed through a message loop—and `false` for a message that has not yet been delivered by any means.

IsSourceRemote() returns `true` if the message had its source in another application, and `false` if the source is local or the message hasn't been delivered yet.

IsSourceWaiting() returns `true` if the message source is waiting for a synchronous reply, and `false` if not. The source thread can request and wait for a reply when calling either BMessenger's **SendMessage()** or BMessage's **SendReply()** function.

IsReply() returns `true` if the BMessage is a reply to a previous message (if it was sent by the **SendReply()** function), and `false` if not.

Previous() returns the previous message—the message to which the current BMessage is a reply. It works only for a BMessage that's received as an asynchronous reply to a previous message. A synchronous reply is received in the context of the previous message, so it's not necessary to call a function to get it. But when an asynchronous reply is received, the context of the original message is lost; this function can provide it. **Previous()** returns `NULL` if the BMessage isn't an asynchronous reply to another message.

See also: **BMessenger::SendMessage()**, **SendReply()**, **ReturnAddress()**

WasDropped() , DropPoint()

bool **WasDropped**(void) const
BPoint **DropPoint**(BPoint *offset* = NULL) const

WasDropped() returns `true` if the user delivered the BMessage by dragging and dropping it, and `false` if the message was posted or sent in application code or if it hasn't yet been delivered at all.

DropPoint() reports the point where the cursor was located when the message was dropped (when the user released the mouse button). It directly returns the point in the screen coordinate system and, if an *offset* argument is provided, returns it by reference in coordinates based on the image or rectangle the user dragged. The *offset* assumes a coordinate system with (0.0, 0.0) at the left top corner of the dragged rectangle or image.

Since any value can be a valid coordinate, **DropPoint()** produces reliable results only if **WasDropped()** returns `true`.

See also: `BView::DragMessage()`

Operators

= (assignment)

```
BMessage &operator =(const BMessage&)
```

Assigns one BMessage object to another. After the assignment, the two objects are duplicates of each other without shared data.

new

```
void *operator new(size_t numBytes)
```

Allocates memory for a BMessage object, or takes the memory from a previously allocated cache. The caching mechanism is an efficient way of managing memory for objects that are created frequently and used for short periods of time, as BMessages typically are.

delete

```
void operator delete(void *memory, size_t numBytes)
```

Frees memory allocated by the BMessage version of new, which may mean restoring the memory to the cache.

The Be Book The Application Kit The Application Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessage Function Summary

Function Summary

Function	Description
=	Provided for implementation
AddBool()	Places boolean in the BMessage
AddData()	Places untyped data in the BMessage
AddDouble()	Places double in the BMessage
AddFlat()	Places BFlattenable in the BMessage
AddFloat()	Places float in the BMessage
AddInt16()	Places int16 in the BMessage
AddInt32()	Places int32 in the BMessage
AddInt64()	Places int64 in the BMessage
AddInt8()	Places int8 in the BMessage
AddMessage()	Places BMessage in the BMessage
AddMessenger()	Places BMessenger in the BMessage
AddPoint()	Places BPoint in the BMessage
AddPointer()	Places pointer in the BMessage
AddRect()	Places BRect in the BMessage
AddRef()	Places entry_ref in the BMessage
AddSpecifier()	Adds a specifier to the specifier stack of a scripting message
AddString()	Places string in the BMessage
CountNames()	Returns the number of named data fields in the object
DropPoint()	Returns the screen location where the message was dropped

FindBool()	Retrieves boolean from the BMessage
FindData()	Retrieves untyped data from the BMessage
FindDouble()	Retrieves double from the BMessage
FindFlat()	Retrieves BFlattenable from the BMessage
FindFloat()	Retrieves float from the BMessage
FindInt16()	Retrieves int16 from the BMessage
FindInt32()	Retrieves int32 from the BMessage
FindInt64()	Retrieves int64 from the BMessage
FindMessage()	Retrieves BMessage from the BMessage
FindMessenger()	Retrieves BMessenger from the BMessage
FindPoint()	Retrieves BPoint from the BMessage
FindPointer()	Retrieves pointer from the BMessage
FindRect()	Retrieves BRect from the BMessage
FindRef()	Retrieves entry_ref from the BMessage
FindString()	Retrieves string from the BMessage
FintInt8()	Retrieves int8 from the BMessage
Flatten()	Writes the object to an untyped buffer of bytes
FlattenedSize()	Returns the number of bytes required to hold the object when flattened
GetCurrentSpecifier()	Retrieves the current specifier from

	the specifier stack
GetInfo()	Provides information about the data fields stored in the BMessage
HasSpecifiers()	Tests whether the object has a specifier stack
IsEmpty()	Tests whether the object has any data
IsReply()	ts whether the object was sent in response to another message
IsSourceRemote()	Tests whether the message originated in the current team
IsSourceWaiting()	TesTests whether the message source is waiting for a synchronous reply
IsSystem()	Tests whether the object is a system-defined message
MakeEmpty()	Removes and frees all data added to the BMessage
BMessage()	Constructor
~BMessage()	Destructor
PopSpecifier()	Retrieves and removes the current specifier from the specifier stack
Previous()	Returns the message to which the current message is a reply
PrintToStream()	Prints information about the BMessage to stdout
RemoveData()	Removes a given item of data from the BMessage
RemoveName()	Removes all data in the BMessage with a given name

ReplaceBool()	Replaces boolean data with the specified name in the BMessage
ReplaceData()	Replaces untyped data with the specified name in the BMessage
ReplaceDouble()	Replaces double data with the specified name in the BMessage
ReplaceFlat()	Replaces BFlattenable data with the specified name in the BMessage
ReplaceFloat()	Replaces float data with the specified name in the BMessage
ReplaceInt16()	Replaces int16 data with the specified name in the BMessage
ReplaceInt32()	Replaces int32 data with the specified name in the BMessage
ReplaceInt64()	Replaces int64 data with the specified name in the BMessage
ReplaceInt8()	Replaces int8 data with the specified name in the BMessage
ReplaceMessage()	Replaces BMessage data with the specified name in the BMessage
ReplaceMessenger()	Replaces BMessenger data with the specified name in the BMessage
ReplacePoint()	Replaces BPoint data with the specified name in the BMessage
ReplacePointer()	Replaces pointer data with the specified name in the BMessage
ReplaceRect()	Replaces BRect data with the specified name in the BMessage

ReplaceRef()	Replaces entry_ref data with the specified name in the BMessage
ReplaceString()	Replaces string data with the specified name in the BMessage
ReturnAddress()	Returns a BMessenger suitable for replying to the BMessage
SendReply()	Sends a reply back to the sender of the BMessage
Unflatten()	Restores the object from an untyped buffer of bytes
WasDelivered()	Tests whether it's possible to send a reply to the message
WasDropped()	Returns whether the object was delivered by the drag and drop mechanism

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Constants and Defined Types

This section lists the constants and types defined by the Support Kit and used throughout the entire Be operating system. Not included here are constants used as **`status_t`** values (error codes). They're listed in "Error Codes".

Constants

Boolean Constants

Declared in: be/support/SupportDefs.h

Constant	Value
false	0
true	1

These constants are defined as values for the **bool** type (described in the next section). The BeOS defines them for C code only. Because they match the boolean symbols that are part of the C++ language, they let you use the same `bool` type and `true` and `false` values when programming in both languages.

See also: **bool**

Byte Swapping Constants

Declared in: be/support/ByteOrder.h

Constant	Description
B_SWAP_HOST_TO_LENDIAN	Convert from host byte order to little endian
B_SWAP_HOST_TO_BENDIAN	Convert from host byte order to big endian
B_SWAP_LENDIAN_TO_HOST	Convert from little endian to host byte order
B_SWAP_BENDIAN_TO_HOST	Convert from big endian to host byte order
B_SWAP_ALWAYS	Always convert endianness

These constants define the action to be taken by the **swap_data()** general byte swapping function.

See also: **swap_data()**

Encoding Conversion Constants

Declared in: be/support/UTF8.h

Constant
B_ISO1_CONVERSION
B_ISO2_CONVERSION
B_ISO3_CONVERSION
B_ISO4_CONVERSION
B_ISO5_CONVERSION
B_ISO6_CONVERSION

B_ISO7_CONVERSION
B_ISO8_CONVERSION
B_ISO9_CONVERSION
B_ISO10_CONVERSION
B_MAC_ROMAN_CONVERSION
B_SJIS_CONVERSION
B_EUC_CONVERSION

These constants identify character encodings to the `convert_to_utf8()` and `convert_from_utf8()` functions, which convert text to and from the standard UTF8 character encoding assumed by the BeOS. The identify the source encoding for a conversion to UTF-8 and the destination encoding for a conversion from UTF-8.

See also: `convert_to_utf8()`, the BFont class in the Interface Kit

Empty String

Declared in: be/support/SupportDefs.h

const char ***B_EMPTY_STRING**

This constant provides a global pointer to an empty string ("").

URL MIME Types

Declared in: be/support/TypeConstants.h

Constant	Meaning
B_URL_HTTP	An HTTP (HyperText Transport Protocol) link
B_URL_HTTPS	An HTTPS (HyperText Transport Protocol Secure) link
B_URL_FTP	AN FTP (File Transfer Protocol) link
B_URL_GOPHER	A Gopher link
B_URL_MAILTO	A mailto (email) link
B_URL_NEWS	A news (usenet) link
B_URL_NNTP	An NNTP (network time protocol) link
B_URL_TELNET	A telnet link
B_URL_RLOGIN	An rlogin link
B_URL_TN3270	A TN3270 link
B_URL_WAIS	A WAIS (Wide Area Information Services) link
B_URL_FILE	A local file link

These are MIME type strings for various URL types. An application can register itself as a handler for one of these types. Once that's done, clicking a link of that type in NetPositive will

cause that application to be launched, with a **B_ARGV_RECEIVED** message with the full URL as the second argument.

NULL and NIL

Declared in: be/support/SupportDefs.h

Constant	Value
NIL	0
NULL	0

These constants represent "empty" values. They're synonyms that can be used interchangeably.

Type Codes

Declared in: be/support/TypeConstants.h

Constant	Meaning
B_CHAR_TYPE	A single character
B_INT8_TYPE	An 8-bit integer
B_INT16_TYPE	A 16-bit integer
B_INT32_TYPE	A 32-bit integer
B_INT64_TYPE	A 64-bit integer
B_UINT8_TYPE	An unsigned 8-bit integer
B_UINT16_TYPE	An unsigned 16-bit integer
B_UINT32_TYPE	An unsigned 32-bit integer
B_UINT64_TYPE	An unsigned 64-bit integer
B_FLOAT_TYPE	A `float`
B_DOUBLE_TYPE	A `double`
B_BOOL_TYPE	A boolean value (the **bool** type)
B_OFF_T_TYPE	An `off_t` value
B_SIZE_T_TYPE	A `size_t` value
B_SSIZE_T_TYPE	An `ssize_t` value
B_POINTER_TYPE	A pointer of some kind (including `void *`)
B_OBJECT_TYPE	An object pointer (such as BMessage *)
B_MESSAGE_TYPE	A BMessage object (not BMessage *)
B_MESSENGER_TYPE	A BMessenger object
B_POINT_TYPE	A BPoint object
B_RECT_TYPE	A BRect object
B_PATH_TYPE	A BPath object
B_REF_TYPE	An **entry_ref** structure

B_RGB_COLOR_TYPE	An **rgb_color** structure
B_PATTERN_TYPE	A **pattern** structure
B_STRING_TYPE	A null-terminated character string
B_MONOCHROME_1_BIT_TYPE	Raw data for a monochrome bitmap (1 bit/pixel)
B_GRAYSCALE_8_BIT_TYPE	Raw data for a grayscale bitmap (8 bits per pixel)
B_COLOR_8_BIT_TYPE	Raw bitmap data in the **B_COLOR_8_BIT** color space
B_RGB_32_BIT_TYPE	Raw bitmap data in the **B_RGB_32_BIT** color space
B_TIME_TYPE	64-bit microsecond **time_t** data
B_MEDIA_PARAMETER_TYPE	A Media Kit parameter.
B_MEDIA_PARAMETER_WEB_T YPE	A Media Kit parameter web.
B_MEDIA_PARAMETER_GROUP _TYPE	A Media Kit parameter group.
B_RAW_TYPE	Raw, untyped data—a stream of bytes
B_MIME_TYPE	The type is specified by a MIME string.
B_ANY_TYPE	The type can be any type.

These constants describe the types of data held by BMessage objects (the Application Kit) and as resources and file system attributes (the Storage Kit). **B_ANY_TYPE** refers to all types; it indicates that the exact type doesn't matter. **B_MIME_TYPE** indicates that the name of the data in the BMessage is a MIME string that specifies its true data type. The other constants refer only to a particular type. The **type_code** defined type marks where these constants are used in the API.

Applications can define their own type codes for data types not found on this list. All the codes the BeOS defines have values formed by concatenating four uppercase letters into a multicharacter constant. For example, **B_MESSENGER_TYPE** is 'MSNG' and **B_SIZE_T_TYPE** is 'SIZT'.

See also: **type_code**

Defined Types

bigtime_t

Declared in: be/support/SupportDefs.h

```
typedef int64 bigtime_t
```

This type records the time in microseconds as a 64-bit integer. Typically, a **bigtime_t** variable measures the system time, the number of microseconds since 12:00:00 AM January 1, 1970, UTC (Coordinated Universal Time).

See also: **system_time()**

bool

Declared in: be/support/SupportDefs.h

```
typedef unsigned char bool
```

The C++ language defines **bool** as its basic boolean type. The BeOS extends the definition to C code, so you can use the same type in both languages. The **true** and **false** constants (listed above) are defined as boolean values.

Function Pointers

Declared in: be/support/SupportDefs.h

```
typedef int (*B_PFI)()
    typedef long (*B_PFL)()
    typedef void (*B_PFV)()
```

These types define pointers to functions that return `int`, `long`, and `void` values respectively.

instantiation_func

Declared in: be/support/Archivable.h

```
typedef BArchivable *(*instantiation_func) (BMessage *)
```

This type is a pointer to a function that can instantiate an object from a BMessage archive and return a pointer to the new object. The **Instantiate()** member function of **BArchivable** is such a function.

See also: the BArchivable class, **find_instantiation_func()**

Integer Types

Declared in: be/support/SupportDefs.h

```
typedef unsigned char uchar

typedef signed char int8
        typedef unsigned char uint8
    typedef volatile signed char vint8
    typedef volatile unsigned char vuint8

typedef short int16
    typedef unsigned short uint16
    typedef volatile short vint16
    typedef volatile unsigned short vuint16

typedef long int32
    typedef unsigned long uint32
    typedef volatile long vint32
    typedef volatile unsigned long vuint32

typedef long long int64
    typedef unsigned long long uint64
    typedef volatile long long vint64
    typedef volatile unsigned long long vuint64
```

These type names are defined as convenient shorthands for standard integers of various sizes. They're used in place of `int`, `short`, and `long` throughout the BeOS API.

The number at the end of the type name indicates the size of the integer. For example, a **`uint32`** is an unsigned 32-bit value. The type is guaranteed to be defined to its stated size for all platforms (thus their exact definitions may vary from platform to platform).

Using these types will make the code you write more portable. They'll help avoid problems as the operating system and your application move to other platforms.

---

status_t

Declared in: be/support/SupportDefs.h

```
typedef int32 status_t
```

This type indicates an error code; it's used mainly for function return values.

See also: "Error Codes"

---

type_code

Declared in: be/support/SupportDefs.h

```
typedef uint32 type_code
```

This type is used for the integer codes that indicate a particular data type. The codes—such as `B_UINT32_TYPE` and `B_MIME_TYPE`—mark the type of data added to a message or stored as a resource and also appear in other contexts.

See also: "Type Codes", the BMessage class in the Application Kit, the BResource class in the Storage Kit

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Functions and Macros

Library: libbe.so
Summary: *more...*

This section lists the Support Kit's general-purpose functions and macros.

---

add_system_beep_event() see beep()

---

atomic_add() , atomic_and() , atomic_or()

Declared in: be/support/SupportDefs.h

```
int32 atomic_add(vint32 *atomicVariable, int32 addValue)
int32 atomic_and(vint32 *atomicVariable, int32 andValue)
int32 atomic_or(vint32 *atomicVariable, int32 orValue)
```

These functions perform the named operations (addition, bitwise *AND*, or bitwise *OR*) on the 32-bit value found in *atomicVariable*, thus:

```
*atomicVariable += addValue
*atomicVariable &= andValue
*atomicVariable |= orValue
```

Each function returns the previous value of the `vint32` variable that *atomicVariable* points to (in other words, they each return the value that was in **atomicVariable* before the operation was performed).

These functions are guaranteed to be *atomic*: if two threads attempt to access the same atomic variable at the same time (through these functions), one of the two threads will be made to wait until the other thread has completed the operation and updated the *atomicVariable* value.

---

beep() , system_beep() , add_system_beep_event()

Declared in: be/support/Beep.h

```
status_t beep(void)
status_t system_beep(const char *eventName)
status_t add_system_beep_event(const char *eventName,
    uint32 flags = _BEEP_FLAGS)
```

beep() produces the basic system beep. This function engages the Media Server, but doesn't wait for the sound to play. If it can't contact the server to play the beep, it returns **B_ERROR**. If it can make contact but can't get a satisfactory reply back from the server, it returns **B_BAD_REPLY**. Otherwise, it returns **B_OK**.

system_beep() plays the system beep configured for the event specified by the given *eventName*.

Applications can add new system beep events by calling **add_system_beep_event()**, passing the name to assign the event; the flags argument is currently unused and should be zero. Once this has been done, the user can use the Sounds preference application to configure a sound for the event.

See also: **play_sound()** in the Media Kit

---

cast_as() see class_name()

---

class_name() , is_instance_of() , is_kind_of() , cast_as()

Declared in: be/support/ClassInfo.h

const char ***class_name**(*object*)*
bool **is_instance_of(***object*, *class***)**
bool **is_kind_of(***object*, *class***)**
class ***cast_as(***object*, *class***)**

These macros deliver information about an object's type, including the name of its class and its standing in the class hierarchy. In each case, the *object* argument is a pointer to an object; it can be an object of any type (it doesn't have to descend from any particular class). The *class* argument is a class name—*not* a string such as "BApplication", but the type name itself (literally `BApplication`).

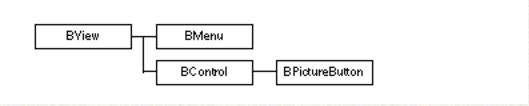
class_name() returns a pointer to the name of *object*'s class.

is_instance_of() returns `true` if *object* is an instance of *class*, and `false` otherwise.

is_kind_of() returns **true** if *object* is an instance of *class* or an instance of any class that inherits from *class*, and **false** if not.

cast_as() returns a pointer to *object* cast as a pointer to an object of *class*, but only if *object* is a kind of *class*. If not, *object* cannot be safely cast as a pointer to *class*, so **cast_as()** returns **NULL**.

For example, given this slice of the inheritance hierarchy from the Interface Kit,



and code like this that creates an instance of the BPictureButton class,

```
BButton *anObject = new BPictureButton(...);
```

the first three macros would work as follows:

- The **class_name()** macro would return a pointer to the string "BPictureButton":

```
const char *s = class_name(anObject);
```

- The **is_instance_of()** macro would return **true** only if the *class* passed to it is BPictureButton. In the following example, it would return **false**, and the message would not be printed. Even though BPictureButton inherits from BView, the object is an instance of the BPictureButton class, not BView:

```
if ( is_instance_of(anObject, BView) )
   printf("The object is an instance of BView.n");
```

- The **is_kind_of()** macro would return **true** if *class* is BPictureButton or any class that BPictureButton inherits from. In the following example, it would return **true** and the message would be printed. A BPictureButton is a kind of BView:

```
if ( is_kind_of(anObject, BView) )
   printf("The object is a kind of BView.n");
```

Note that class names are not passed as strings, but **class_name()** returns the name as a string.

The **cast_as()** macro is most useful when you want to treat a generic object as an instance of a more specific class. Suppose, for example, that the BPictureButton mentioned above becomes the focus view for a window and you retrieve it by calling the BWindow's **CurrentFocus()** function:

```
BView *focus = myWindow->CurrentFocus();
```

Since the focus view might be any type of view, **CurrentFocus()** returns a pointer to an object of the base BView class. Unless you know otherwise, you cannot treat the object as anything more specific than a BView instance. However, you can ask the object if it's a kind of BPictureButton and, if it is, cast it to the BPictureButton type:

```
if ( is_kind_of(focus, BPictureButton) ) {
   BPictureButton *picbutton = (BPictureButton *)focus);
   if ( picbutton->Behavior() == B_TWO_STATE_BUTTON )
```

```
      . . .
}
```

The **cast_as()** macro does the same thing, but more efficiently. It casts the object to the target class if it is safe to do so—if the *object* is an instance of a class that inherits from the target *class* or an instance of the target *class* itself—and returns `NULL` if not.

```
BPictureButton *picbutton = cast_as(focus, BPictureButton);
if ( picbutton ) {
   if ( picbutton->Behavior() == B_TWO_STATE_BUTTON )
      . . .
}
```

cast_as() is often used in place of the cast operator to assure code safety even where an expected result is anticipated and there's no need for an intermediate variable (like *focus*):

```
BPictureButton *picbutton =
            cast_as(myWindow->CurrentFocus(), BPictureButton);
if ( picbutton ) {
      . . .
}
```

The **cast_as()** and **is_kind_of()** macros work alike; they're both based on the C++ **dynamic_cast** operator and they reflect its behavior. To describe that behavior more precisely, let's adopt the following shorthand terms for an object's type:

- The *real type* of an object is its type on construction. For example, if you construct an instance of the BButton class, as shown above, BButton is its real type.
- The *declared type* of an object is the class label it currently bears. For example, **CurrentFocus()** returns an object whose declared class is BView.

Either of these types can be compared to a *target type*, the type you want to cast the object to or test it against. The target type is the *class* argument passed to the macros.

In the best of all possible worlds, you'd want to ignore the declared type of an object and compare only the real type to the target type. However, the `dynamic_cast` operator—and by extension **cast_as()** and **is_kind_of()**—considers the real type only if it has to. It first compares the object's declared type to the target type. It assumes that the declared type is accurate (that the object is truly the kind of object it's represented to be) and it summarily handles the obvious cases: If the target type is the same as the declared type or if it's a class that the declared type inherits from, the operation will succeed. Consequently, **cast_as()** will cast the object to the target type and **is_kind_of()** will return `true`, regardless of the object's real type. In other words, if the target class is above or at the same level as the declared class in the inheritance hierarchy, the real class is ignored.

However, if the declared type doesn't match or derive from the target type, `dynamic_cast` and the macros look at the real type: If the target class is identical to the real type, or if it's a class

that the real type derives from, the operation succeeds. If not, it fails.

Therefore, the **is_kind_of()** and **cast_as()** macros will produce reliable results as long as objects are not arbitrarily cast to types that may not be accurate. For example, you should not cast an object to a target type and then attempt to use **is_kind_of()** to determine if the cast was correct. This code is unreliable:

```
BPictureButton *picbutton =
                  (BPictureButton *)myWindow->CurrentFocus();
if ( is_kind_of(picbutton, BPictureButton) ) {
      . . .
}
```

In this example, **is_kind_of()** will always return `true`, no matter what the class of the current focus view. The general rule is that the declared type of an object must *always* be accurate; an object should be typed only to its own class or to a class that it inherits from. The macros cannot rescue you from an inaccurate cast.

---

convert_to_utf8() , convert_from_utf8()

Declared in: be/support/UTF8.h

```
status_t convert_to_utf8(uint32 sourceEncoding,
    const char *source,
    int32 *sourceLength,
    char *destination,
    int32 *destinationLength,
    int32 *state,
    char substitute = B_SUBSTITUTE)
```

```
status_t convert_from_utf8(uint32 destinationEncoding,
    const char *source,
    int32 *sourceLength,
    char *destination,
    int32 *destinationLength,
    int32 *state,
    char substitute = B_SUBSTITUTE)
```

These functions convert text to and from the Unicode UTF-8 encoding that's standard for the Be operating system and is assumed in most contexts. UTF-8 is described under "Character Encoding" in "Responding to the User" section of **The Interface Kit** chapter.

convert_to_utf8() permits you to take text that's encoded according to another standard and convert it to UTF-8 for the BeOS. **convert_from_utf8()** lets you convert text from UTF-8 to other encodings for other venues (for example, to the encodings commonly used for displaying text on the World Wide Web).

The first argument passed to these functions names the other encoding—the source encoding for `convert_to_utf8()` and the destination encoding for `convert_from_utf8()`. It can be any of the following constants:

`B_ISO1_CONVERSION`
`B_ISO2_CONVERSION`
`B_ISO3_CONVERSION`
`B_ISO4_CONVERSION`
`B_ISO5_CONVERSION`
`B_ISO6_CONVERSION`
`B_ISO7_CONVERSION`
`B_ISO8_CONVERSION`
`B_ISO9_CONVERSION`
`B_ISO10_CONVERSION`
`B_MAC_ROMAN_CONVERSION`
`B_SJIS_CONVERSION`
`B_EUC_CONVERSION`
`B_JIS_CONVERSION`
`B_MS_WINDOWS_CONVERSION`
`B_UNICODE_CONVERSION`
`B_KOI8R_CONVERSION`
`B_MS_WINDOWS_1251_CONVERSION`
`B_MS_DOS_866_CONVERSION`

Most of these constants designate encoding schemes that are supported by the BFont class in the Interface Kit and its `SetEncoding()` function. They parallel the constants that are passed to that function. For example, `B_ISO1_CONVERSION` (for these functions) and `B_ISO_8859_1` (for `SetEncoding()`) both designate the extended ASCII encoding defined in part one of ISO 8859 (Latin 1). Similarly, `B_ISO2_CONVERSION` matches `B_ISO_8859_2`, `B_ISO3_CONVERSION` matches `B_ISO_8859_3`, and so on. `B_MAC_ROMAN_CONVERSION` matches `B_MACINTOSH_ROMAN`. (`B_ISO10_CONVERSION` is not implemented for this release.)

`B_SJIS_CONVERSION` stands for the Shift-JIS (Japanese Industrial Standard) encoding of Japanese and `B_EUC_CONVERSION` stands for the EUC (Extended UNIX Code) encoding of Japanese in packed format.

Both functions convert up to *sourceLength* bytes of text from the *source* buffer. They write up to *destinationLength* bytes of converted text into the *destination* buffer. The amount of text that they actually convert is therefore constrained both by the amount of source text (*sourceLength*) and the capacity of the output buffer (*destinationLength*). Neither function stops at a null terminator ('0') when reading the input buffer nor adds one to the text in the output buffer; they depend only on *sourceLength* and *destinationLength* for guidance.

When finished, these functions modify the variable that *sourceLength* refers to so that it reports the number of bytes of source text actually converted. They also modify the variable that *destinationLength* refers to so that it reports the number of bytes actually written to the destination buffer. Neither function will stop in the middle of a multibyte source character; they're guaranteed to convert only full characters.

The *state* argument serves as a cookie that lets you use multiple calls to these functions to convert a batch of text (if, for example, the text is stored in multiple buffers). Pass 0 for the first buffer, and pass the value returned in *state* to the next call to continue processing the same text; this will cause the conversion to continue where it left off. For example:

```
/* buffer is a pointer to a small source buffer */
/* destBuffer is a pointer to a small destination buffer */
/* destBufferLen is the size of the destination buffer in bytes */
int32 state = 0;
int32 bufferLen;

while ((bufferLen = GetNextBuffer(file, buffer))) {
   convert_to_utf8(buffer, &bufferLen, destBuffer, &destBufferLen, &state);
   /* do stuff with the buffer in destBuffer, which has the converted text */
}
```

In this example, text is fetched using a call to a function called **`GetNextBuffer()`**, whose implementation depends on the application's needs. Each buffer is converted, and the converted text is stored in `destBuffer`, which can then be used by the application.

If either function encounters a character in the source that the destination format doesn't allow, it puts the character specified by the *substitute* argument (or a question mark ('?') if *substitute* isn't specified) in its place in the output text. This is much more likely to occur when converting from UTF-8 than when converting to it, since Unicode represents a very large number of characters.

If successful in converting at least one source character, both functions return **`B_OK`**. If unsuccessful, for example, if they don't recognize the source or destination encoding, they return **`B_ERROR`**. If there's an error, you should not trust any of the output arguments.

These functions are found in the **libtextencoding.so** library. If you use them, be sure to include this library file.

See also: **`BFont::SetEncoding()`**, "Character Encoding" in the "Responding to the User" section of the **The Interface Kit** chapter

---

find_instantiation_func()

Declared in: be/support/Archivable.h

```
instantiation_func find_instantiation_func(const char *className)
```

instantiation_func **find_instantiation_func(**BMessage ***archive*)**

Returns a pointer to the **Instantiate()** function that can create instances of the *className* class, or NULL if the function can't be found. If passed a BMessage *archive*, **find_instantiation_func()** gets the name of the class from a **B_STRING_TYPE** field called "class" in the BMessage.

The **instantiation_func** type is defined as follows:

```
BArchivable *(*instantiation_func) (BMessage *)
```

In other words, the function has the same syntax as the **Instantiate()** function declared in the BArachivable class and replicated in derived classes (with class-specific return values).

The function that's returned can be called like any C function; you don't need the class name or another object of the class. For example:

```
instantiation_func func;
if ( func = find_instantiation_func(arhiveMessage) ) {
   BArchivable *object = func(archiveMessage);
}
```

instantiate_object() will do this work for you.

See also: **BArchivable::Instantiate()**, **instantiate_object()**

---

instantiate_object()

Declared in: be/support/Archivable.h

BArchivable ***instantiate_object(**BMessage ***archive*)**

Creates and returns a new instance of an archived object, or returns NULL if the object can't be constructed. The object is created by calling the **Instantiate()** function of...

1. ...the class that was last added to the "class" field (an array of class names) of the *archive* BMessage. This will be the "most derived" class in the array.

2. If the named class isn't recognized by the app (or doesn't define **Instantiate()**), the function tries to load the add-on that's identified (by signature) in *archive*'s "add_on" field. This add-on should contain the code for the class.

3. If the add-on method doesn't work, **instantiate_object()** tries the next class in the "class" array, and so works its way up the class hierarchy.

4. If the appropriate class is never found, the function returns NULL.

When successful, **instantiate_object()** returns the object that **Instantiate()** created, but typed to the base BArchivable class. The **cast_as()** macro can type it to a the proper class.

```
BArchivable *base = instantiate_object(archive);
if ( base ) {
   TheClass *object = cast_as(base, TheClass);
   if ( object ) {
      . . .
   }
}
```

See also: the BArchivable class, **find_instantiation_func()**

- is_instance_of() see class_name()
- is_kind_of() see class_name()
- is_type_swapped() see swap_data()

- swap_data() , is_type_swapped()

Declared in: be/support/ByteSwap.h

status_t **swap_data**(type_code *type*, void **data*, size_t *length*, swap_action *action*)
bool **is_type_swapped**(type_code *type*)

`swap_data` is a general byte swapping function, converting *data* of type *type* and converting its endianness according to *action* (defined in "Byte Swapping Constants"). The *length* field can be used to specify an array of items to be converted. For example, you can convert an array of BMessengers from little endian to the host endianness:

```
BMessenger messengers[10];
...
swap_data(B_MESSENGER_TYPE, messengers, 10*sizeof(BMessengers)
         B_SWAP_LENDIAN_TO_HOST);
```

The function can swap most data types with type constants defined in **<be/support/TypeConstants.h>**. It returns `B_OK` on success and `B_BAD_VALUE` on failure.

`is_type_swapped` takes a type code and determines whether or not it has been swapped. This only works for types defined in **<be/support/TypeConstants.h>**. It returns `true` if the type code is in the host's native endianness and `false` otherwise.

See also: "Byte Swapping Constants"

- min() , max() , min_c() , max_c()

Declared in: be/support/SupportDefs.h

min(*a*, *b*)
min_c(*a*, *b*)
max(*a*, *b*)
max_c(*a*, *b*)

These macros compare two integers or floating-point numbers. **min()** and **min_c()** return the lesser of the two (or *b* if they're equal); **max()** and **max_c()** return the greater of the two (or *a* if they're equal). **min()** and **max()** can only be used in straight C programs. **min_c()** and **max_c()** are defined for C *and* C++.

---

system_ see beep()

---

validate_instantiation()

Declared in: be/support/Archivable.h

bool **validate_instantiation**(BMessage **archive*, const char **className*)

Returns `true` if the *archive* BMessage contains data for an object belonging to the *className* class, and `false` if not. The determination is made by looking for the class name in a "class" array in the *archive*. If the class name appears anywhere in the array, this function returns `true`. If not, it returns **`false`**.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Functions and Macros Function Summary

Function Summary

Function	Description
add_system_beep_event()	Adds a system beep event
atomic_add()	Atomic addition of variables
atomic_and()	Atomic and of variables
atomic_or()	Atomic or of variables
beep()	Produces the system beep
cast_as()	Safe casting operation
class_name()	Returns the name of the class
convert_from_utf8()	Convert text from UTF8
convert_to_utf8()	Convert text to UTF8
find_instantiation_func()	Returns a pointer to an object's **`Instantiate()`** function
instantiate_object()	Instantiate an archived object
is_instance_of()	Tests whether an object is an instance of a given class
is_kind_of()	Tests whether an object is an instance of or inherits from a given class
is_type_swapped()	Tests whether a type code is swapped
max()	Compares two values, returning the greater
max_c()	Compares two values, returning the greater

min()	Compares two values, returning the lesser
min_c()	Compares two values, returning the lesser
swap_data()	General byte swapping function
system_beep()	Produces the system beep for a specific event
validate_instantiation()	Tests whether an archive contains an object of the given type

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Functions

This section lists Media Kit functions that aren't contained in classes.

Global Functions

format_is_compatible()

Declared in: be/media/MediaDefs.h

```
bool format_is_compatible(const media_format &format1,
    const media_format &format2)
```

Returns **true** if media data of format *format1* can be fed into a consumer that accepts data in media_format *format2*; otherwise, returns **false**.

get_next_file_format()

Declared in: be/media/MediaDefs.h

```
status_t get_next_file_format(int32 *cookie, media_file_format *info)
```

If you need to search through all available file formats, you can do so using this function. Point *cookie* at an int32 containing 1, and *info* at a **media_file_format** structure, and call `get_next_file_format()`; on return, *info* will contain information about the first available file format.

You can call `get_next_file_format()` repeatedly until **B_BAD_INDEX** is returned, which indicates that there aren't any more file formats to scan.

The following snippet dumps a list of all the file formats, including all the audio and video codecs each file format supports.

```
void dump_info(void) {
   int32 cookie = 0, cookie2;
   media_format format, outfmt;
   media_file_format mfi;
   media_codec_info mci;

   while(get_next_file_format(&cookie, &mfi) == B_OK) {
      printf("%s (%s, id %d)n", mfi.pretty_name,
               mfi.short_name, mfi.id);

      cookie2 = 0;
      memset(&format, 0, sizeof(format));
      format.type = B_MEDIA_RAW_VIDEO;

      format.u.raw_video.last_active = (uint32) (320 - 1);
      format.u.raw_video.orientation = B_VIDEO_TOP_LEFT_RIGHT;
```

```
      format.u.raw_video.pixel_width_aspect = 1;
      format.u.raw_video.pixel_height_aspect = 3;
      format.u.raw_video.display.format = B_RGB32;
      format.u.raw_video.display.line_width = (int32) 320;
      format.u.raw_video.display.line_count = (int32) 240;
      format.u.raw_video.display.bytes_per_row = 4 * 320;

      printf(" Video Encoders:n");
      while (get_next_encoder(&cookie2, &mfi, &format, &outfmt, &mci) == B_OK) {
         printf(" %s / %s (%d)n", mci.pretty_name, mci.short_name,
                  mci.id);
      }

      cookie2 = 0;
      format.type = B_MEDIA_RAW_AUDIO;
      format.u.raw_audio.format = media_raw_audio_format::B_AUDIO_UCHAR;
      format.u.raw_audio.channel_count = 1;
      printf(" Audio Encoders:n");
      while (get_next_encoder(&cookie2, &mfi, &format, &outfmt, &mci) == B_OK) {
         printf(" %s / %s (%d)n", mci.pretty_name, mci.short_name,
                  mci.id);
      }
   }
}
```

RETURN CODES

- B_OK. A format was returned.
- **B_BAD_VALUE**. An invalid *cookie* was specified.
- **B_BAD_INDEX**. No more file formats, or the cookie was invalid.

---

launch_media_server() , shutdown_media_server()

Declared in: be/media/MediaDefs.h

```
status_t launch_media_server(uint32 flags = 0)
status_t shutdown_media_server(bigtime_t timeout = B_INFINITE_TIMEOUT,
    bool (*progress)(int stage, const char *message, void *cookie) = NULL,
    void *cookie = NULL)
```

launch_media_server() starts up the Media Servers (and the old Audio Server); you can use this if you discover that the Media Server isn't started but you need to use it.

`shutdown_media_server()` shuts down the Media Servers (and the old Audio Server). If you'd like to be able to keep track of the shutdown process, specify a pointer to a function in *progress*; this function will receive as input an integer value, *stage*, indicating how far through the shutdown process things are (100 means the shutdown is complete), *message* will be a test string describing what's going on, and *cookie* will be the same value as the cookie you specified as input into `shutdown_media_server()`.

The shutdown process will abort if the *timeout* expires.

---

media_realtime_init_image()

Declared in: be/media/MediaDefs.h

```
status_t media_realtime_init_image(image_id image, uint32 flags)
```

Given the specified *image*, this function prepares the image for realtime media. The *flags* indicate what type of media the image will be handling; this must match the realtime flags specified by the `BMediaRoster::SetRealtimeFlags()` function (or by the Media preference application, which calls it). If the flags don't match up, **B_MEDIA_REALTIME_DISABLED** is returned.

RETURN CODES

- B_OK. The image is ready for realtime media.
- **B_MEDIA_REALTIME_DISABLED**. The media type indicated by flags isn't enabled for realtime (see `BMediaRoster::SetRealtimeFlags()`).
- **B_MEDIA_REALTIME_UNAVAILABLE**. There aren't enough system resources available to enable realtime performance.

---

media_realtime_init_thread()

Declared in: be/media/MediaDefs.h

```
status_t media_realtime_init_thread(thread_id thread, size_t stackUsed, uint32 flags)
```

Prepares the specified *thread* for realtime media. The *stackUsed* argument specifies how much of the stack should be locked down in memory. It's very unlikely that your thread actually uses the entire 256kB stack allocated for it, so you can preserve valuable system resources by indicating an upper bound for the amount of stack to lock in memory. If you need to lock down the entire stack, you can pass 0.

The *flags* argument indicates the type of media the thread will be handling.

RETURN CODES

- B_OK. The image is ready for realtime media.

- **B_MEDIA_REALTIME_DISABLED**. The media type indicated by flags isn't enabled for realtime (see **BMediaRoster::SetRealtimeFlags()**).
- **B_MEDIA_REALTIME_UNAVAILABLE**. There aren't enough system resources available to enable realtime performance.

---

play_sound()

Declared in: be/media/PlaySound.h

```
sound_handle play_sound(const entry_ref *soundRef,
    bool willMix,
    bool willWait,
    bool background)
```

play_sound plays the sound file identified by the **entry_ref** pointed to by the *soundRef* parameter. The *willMix* and *willWait* arguments are used to determine how the function behaves with regard to other sounds.

- If you want the sound to play all by itself, set *willMix* to **FALSE**. If you don't care if it gets mixed with other sounds, set it to **TRUE**.
- If you want your sound to play immediately (whether or not you're willing to mix), set *willWait* to **FALSE**. If you're willing to wait for the sound playback resources to become available, set *willWait* to **TRUE**.

Note that setting *willMix* to **TRUE** doesn't guarantee that your sound will play immediately. If the sound playback resources are claimed for exclusive access by some other process, you'll be blocked, even if you're willing to mix.

The background argument, if **TRUE**, tells the function to spawn a thread in which to play the sound. In this case, the function returns immediately. If background is **FALSE**, the sound is played synchronously and play_sound() won't return until the sound has finished.

The **sound_handle** value that's returned is a token that represents the sound that's being played back, and is only valid if you're playing in the background. You would use this token in a subsequent call to stop_sound() or wait_for_sound(). If the **entry_ref** doesn't represent a file, or if the sound couldn't be played, for whatever reason, play_sound() returns a negative integer.

---

shutdown_media_server() see launch_media_server()

---

stop_sound()

Declared in: be/media/PlaySound.h

```
status_t stop_sound(sound_handle handle)
```

`stop_sound()` stops the playback of a sound identified by *handle*, a value that was returned by a previous call to **`play_sound()`**. The return value can be ignored.

---

string_for_format()

Declared in: be/media/MediaDefs.h

```
bool string_for_format(const media_format &format,
    char *outBuffer,
    size_t bufferSize)
```

Fills the buffer specified by *outBuffer* with a string describing the specified *format*. The buffer's size is specified by the *bufferSize* argument. The string may not be pretty, but it will list all kinds of interesting information about the format.

If a description is returned, this function returns **`true`**. Otherwise, the result is **`false`**.

---

wait_for_sound()

Derived from: be/media/PlaySound.h

```
status_t wait_for_sound(sound_handle handle)
```

Causes the calling thread to block until the sound specified by *handle* has finished playing. The *handle* value should be a value returned by a previous call to **`play_sound()`**. `wait_for_sound()` currently always returns **`B_OK`**.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Constants and Defined Types

This section describes the constants and defined types that make the Media Kit tick.

Constants

Codec Type Info

Declared in: be/media/MediaDefs.h

Constant	Description
B_CODEC_TYPE_INFO	Data is in codec type info format.

The **B_CODEC_TYPE_INFO** constant is used to indicate that data is a codec info structure; it's used in the **user_data_type** field in the **media_format** structure, for example.

Data Endianness

Declared in: be/media/MediaDefs.h

Constant	Description
B_MEDIA_BIG_ENDIAN	The media data is big-endian
B_MEDIA_LITTLE_ENDIAN	The media data is little-endian.

These values indicate whether media data is little-endian or big-endian.

In addition, **B_MEDIA_HOST_ENDIAN** is defined to be one of these values, indicating the endianness of the system on which your application is running.

Media Message Size

Declared in: be/media/MediaDefs.h

```
const size_t B_MEDIA_MESSAGE_SIZE = 16384;
```

The **B_MEDIA_MESSAGE_SIZE** constant specifies the maximum possible size a message a node's service thread can receive. 16k is an upper limit and won't be exceeded. Your thread can simply create a buffer this large, **read_port()** into it, and call **HandleMessage()** to process it.

media_display_flags

Declared in: be/media/MediaDefs.h

Constant	Description
B_F1_DOMINANT	The first buffer sent (temporally) will be an F1 field.
B_F2_DOMINANT	The first buffer sent (temporally) will be an F2 field.
B_TOP_SCANLINE_F1	The topmost scanline of the output buffer belongs to F1.
B_TOP_SCANLINE_F2	The topmost scanline of the output buffer belongs to F2.

These values are used to define the order in which the F1 and F2 fields of video data are used.

media_flags

Declared in: be/media/MediaDefs.h

Constant	Description
`B_MEDIA_FLAGS_VERSION`	Returns the Media Kit version as an int32 value.

The media_flags indicate values you can pass to `BMediaRoster::MediaFlags()` to get information about the version of the Media Kit that your application is running under.

media_format_family

Declared in: be/media/MediaDefs.h

Constant	Meaning
`B_ANY_FORMAT_FAMILY`	Any family.
`B_BEOS_FORMAT_FAMILY`	BeOS format.
`B_QUICKTIME_FORMAT_FAMILY`	QuickTime format.
`B_AVI_FORMAT_FAMILY`	AVI format.
`B_ASF_FORMAT_FAMILY`	ASF format.
`B_MPEG_FORMAT_FAMILY`	MPEG format.
`B_WAV_FORMAT_FAMILY`	WAVE format audio.
`B_AIFF_FORMAT_FAMILY`	AIFF format audio.
`B_AVR_FORMAT_FAMILY`	AVR format audio.
`B_MISC_FORMAT_FAMILY`	Miscellaneous.

These constants indicate the media format families the BMediaFormats class can map among.

media_format_flags

Declared in: be/media/MediaDefs.h

Constant	Description
`B_MEDIA_RETAINED_DATA`	Data in the buffer should be retained between calls to the decoder. A decoder that requires this behavior will set this bit in *require_flags*, and an application that doesn't support this behavior will set this bit in *deny_flags*.
`B_MEDIA_MULTIPLE_BUFFERS`	An application will set this flag in *require_flags* if it passes multiple buffers to the decoder; a decoder that doesn't support this will set this flag in *deny_flags*.

B_MEDIA_CONTIGUOUS_BUFFER	A decoder will set this flag in *require_flags* if it requires that buffers be in contiguous physical memory (for example, if it uses a hardware decoder). An application at doesn't support this should set this bit in *deny_flags*.
B_MEDIA_LINEAR_UPDATES	An application sets this flag in *require_flags* if it requires the decoder to write linearly (instead of randomly) into the buffer (for example, if using overlays). A decoder that really wants to write randomly should set this flag in *deny_flags*.
B_MEDIA_MAUI_UNDEFINED_FLAGS	Both applications and decoders should set this in both *require_flags* and *deny_flags* to avoid allowing or denying support for features they know nothing about.

These flags are used when specifying media_format's *allow_flags* or *deny_flags* fields, to indicate how the decoder should behave.

---

media_multi_matrix

Declared in: be/media/MediaDefs.h

Constant	Description
B_MATRIX_PROLOGIC_LR	Dolby ProLogic matrix.
B_MATRIX_AMBISONIC_WXYZ	Ambisonic WXYZ matrix.

These values identify a particular multi-channel audio matrix.

---

media_producer_status

Declared in: be/media/MediaDefs.h

Constant	Description
B_DATA_NOT_AVAILABLE	There aren't any buffers ready for the destination.
B_DATA_AVAILABLE	There are buffers ready for the destination.
B_PRODUCER_STOPPED	The producer isn't running.

These values can be specified when your producer node calls **BBufferProducer::SendDataStatus()** to tell the consumers it's feeding whether or not data is available for it to process. The **B_PRODUCER_STOPPED** code is sent to indicate to downstream consumers that it's no longer running.

Consumers receive these values in the **BBufferConsumer::ProducerDataStatus()** hook function.

---

media_realtime_flags

Declared in: be/media/MediaDefs.h

Constant	Description
`B_MEDIA_REALTIME_ALLOCATOR`	When set, rtm_alloc() will return locked memory.
`B_MEDIA_REALTIME_AUDIO`	Audio add-ons in the Media Server are locked in memory, and should lock their thread stacks using `media_init_realtime_thread()`.
`B_MEDIA_REALTIME_VIDEO`	Video add-ons are locked in memory, and should lock their thread stacks using `media_init_realtime_thread()`.
`B_MEDIA_REALTIME_ANYKIND`	All Media add-ons are locked in memory, and should lock their thread stacks using `media_init_realtime_thread()`.

These values are passed to realtime allocator functions to indicate what types of nodes should use locked memory.

---

media_type

Declared in: be/media/MediaNode.h

Constant	Description
`B_MEDIA_NO_TYPE`	Typeless media data.
`B_MEDIA_UNKNOWN_TYPE`	Unknown media type.
`B_MEDIA_RAW_AUDIO`	Uncompressed, linear audio.
`B_MEDIA_RAW_VIDEO`	Uncompressed, linear video.
`B_MEDIA_VBL`	Raw data from VBL area.
`B_MEDIA_TIMECODE`	Timecode data. Undetermined format.
`B_MEDIA_MIDI`	MIDI.
`B_MEDIA_TEXT`	Text data; typically closed-captioning information.
`B_MEDIA_HTML`	HTML data.
`B_MEDIA_MULTISTREAM`	Multi-stream media data, such as AVI or QuickTime.
`B_MEDIA_PARAMETERS`	BControllable change data.
`B_MEDIA_ENCODED_AUDIO`	Encoded audio (such as AC3 or DTS).
`B_MEDIA_ENCODED_VIDEO`	Encoded video (such as Indeo or MPEG).
`B_MEDIA_PRIVATE`	Base value for private Be media types.
`B_MEDIA_FIRST_USER_TYPE`	Base value for user-defined media types.

Types of media data.

---

multi_media_channels

Declared in: be/media/MediaDefs.h

Constant	Description
`B_CHANNEL_LEFT`	The left channel.

B_CHANNEL_RIGHT	The right channel.
B_CHANNEL_CENTER	The center channel (5.1+ or fake surround)
B_CHANNEL_SUB	Subwoofer (5.1+).
B_CHANNEL_REARLEFT	Quad surround or 5.1+.
B_CHANNEL_REARRIGHT	Quad surround or 5.1+
B_CHANNEL_FRONT_LEFT_CENTER	Front-left center channel.
B_CHANNEL_FRONT_RIGHT_CENTER	Front-right center channel.
B_CHANNEL_BACK_CENTER	The rear center channel (6.1 or fake surround).
B_CHANNEL_SIDE_LEFT	The left side channel.
B_CHANNEL_SIDE_RIGHT	The right side channel.
B_CHANNEL_TOP_CENTER	The top center channel.
B_CHANNEL_TOP_FRONT_LEFT	The top front-left channel.
B_CHANNEL_TOP_FRONT_CENTER	The top front-center channel.
B_CHANNEL_TOP_FRONT_RIGHT	The top front-right channel.
B_CHANNEL_TOP_BACK_LEFT	The top back-left channel.
B_CHANNEL_TOP_BACK_CENTER	The top back-center channel.
B_CHANNEL_TOP_BACK_RIGHT	The top back-right channel.

Speaker positions for multi-channel audio.

---

node_kind

Declared in: be/media/MediaNode.h

Constant	Description
B_BUFFER_PRODUCER	The flavor implements BBufferProducer.
B_BUFFER_CONSUMER	The flavor implements BBufferConsumer.
B_TIME_SOURCE	The flavor implements BTimeSource.
B_CONTROLLABLE	The flavor implements BControllable.
B_FILE_INTERFACE	The flavor implements BFileInterface.
B_ENTITY_INTERFACE	The flavor implements BEntityInterface (not available for R4).
B_PHYSICAL_INPUT	The flavor represents a physical input point (such as an input jack).
B_PHYSICAL_OUTPUT	The flavor represents a physical output point (such as line output).
B_SYSTEM_MIXER	The flavor represents the system mixer.

Defines the type or types of node that a particular flavor supports. Note that a flavor can implement more than one type of node, so you can combine these values using a bitwise or.

Quality

Declared in: be/media/MediaDefs.h

Constant	Description
B_MEDIA_ANY_QUALITY	Any quality level (0.0).
B_MEDIA_LOW_QUALITY	Low quality (0.1).
B_MEDIA_MEDIUM_QUALITY	Medium quality (0.5).
B_MEDIA_HIGH_QUALITY	High quality (1.0).

These are default values you can use for various quality levels.

---

video_orientation

Declared in: be/media/MediaNode.h

Constant	Description
B_VIDEO_TOP_LEFT_RIGHT	Typical progressive scan: top to bottom, left to right.
B_VIDEO_BOTTOM_LEFT_RIGHT	Bottom to top, left to right (BMP and TGA graphics are oriented this way).

Indicates the orientation of video data (whether the data is oriented top to bottom or bottom to top).

Defined Types

encode_parameters

Declared in: be/media/MediaDefs.h

```
struct encode_parameters {
    float quality;
    int32 avg_field_size;
    int32 max_field_size;
    int32 _pad[27];
    void *user_data;
    size_t user_data_size;
    };
```

Parameters for encoding a buffer.

- *quality* indicates the compression quality to use. 0.0 is lowest quality, 1.0 is high quality. You can use the `B_MEDIA_ANY_QUALITY`, `B_MEDIA_LOW_QUALITY`, and so forth constants if you wish.
- *avg_field_size* indicates the average size of a field that should be allowed, in bytes. The encoder will try to maintain this average as it compresses.
- *max_field_size* indicates the maximum field size that should be allowed, in bytes. The encoder will try to encode each field to be smaller than this size.
- *user_data* and *user_data_size* indicate user-defined data.

live_node_info

Declared in: be/media/MediaDefs.h

```
struct live_node_info {
    live_node_info();
    ~live_node_info();
    media_node node;
    BPoint hint_point;
    char name[B_MEDIA_NAME_LENGTH];
    private:
    reserved[160];
    };
```

The live_node_info structure describes an active node, and is returned by `BMediaRoster::GetLiveNodes()`.

- *node* specifies which node the live_node_info structure describes.

- *hint_point* is reserved for future use. Pass a BPoint with x and y both set to 0.

---

media_addon_id

Declared in: be/media/MediaDefs.h

```
typdef int32 media_addon_id;
```

A media add-on ID number.

---

media_audio_header

Declared in: be/media/MediaDefs.h

```
struct media_audio_header {
        int32 _reserved_[16];
    };
```

Defines the header of an audio buffer. There are no Be-defined fields in this structure as of the release of BeOS R4; however, you may append fields to the end of this structure if you wish.

---

media_buffer_id

Declared in: be/media/MediaDefs.h

```
typdef int32 media_buffer_id;
```

A media buffer ID number.

---

media_decode_info

Declared in: be/media/MediaDefs.h

```
struct media_decode_info {
    bigtime_t time_to_decode;
    int32 _pad[26];
    void *file_format_data;
    size_t file_format_data_size;
    void *codec_data
        size_t codec_data_size;

    media_decode_info();
    };
```

Provides options to decoders.

- *time_to_decode* indicates how long, in microseconds, the decoder is allowed to take to decode the buffer. Specify 0 to do it as fast as possible, or `B_INFINITE_TIMEOUT` if you don't

care.

- *file_format_data* is a pointer to file-format specific data, and *file_format_data_size* indicates the size of this buffer.
- *codec_data* is a pointer to a codec-specific data buffer, and *codec_data_size* indicates the size of the buffer.

---

media_destination

Declared in: be/media/MediaDefs.h

```
struct media_destination {
    media_destination(port_id, int32);
    media_destination(const media_destination &clone);
    media_destination & operator=  (const media_destination &clone);
    media_destination();
    ~media_destination();
    port_id port;
    int32 id;
    static media_destination null;
    private:
      uint32 _reserved_media_destination_[2];
    };
```

Describes the destination end of a media connection between two nodes. The *port* indicates the port ID on which the destination communicates. This can be different from the media_node.port value.

---

media_encode_info

Declared in: be/media/MediaDefs.h

```
struct media_encode_info {
    uint32 flags;
    int32 used_data_size;
    bigtime_t start_time;
    bigtime_t time_to_encode;
    int32 _pad[22];
    void *file_format_data;
    size_t file_format_data_size;
    void *codec_data
        size_t codec_data_size;

    media_encode_info();
    };
```

Provides options to encoders and file writers about.

- *flags* contains flags that alter the behavior of the encoder or writer. The only defined value right now is **B_MEDIA_KEY_FRAME**.
- *used_data_size* is currently reserved for use by Be.
- *start_time* indicates the time at which the buffer being written is located, in microseconds, from the start of the file.
- *time_to_encode* indicates how long, in microseconds, the encoder is allowed to take to encode the buffer. Specify 0 to do it as fast as possible, or **B_INFINITE_TIMEOUT** if you don't care.
- *file_format_data* is a pointer to file-format specific data, and *file_format_data_size* indicates the size of this buffer.
- *codec_data* is a pointer to a codec-specific data buffer, and *codec_data_size* indicates the size of the buffer.

---

media_encoded_audio_format

Declared in: be/media/MediaDefs.h

```
struct media_encoded_audio_format {
    enum audio_encoding {
      B_ANY,
    };
    media_raw_audio_format output;
    audio_encoding encoding;
    float bit_rate;
    size_t frame_size;
    media_multi_audio_info multi_info;
        uint32  _reserved_[3];
    static media_encoded_audio_format wildcard;
    };
```

Describes the format of encoded audio data.

- *output* indicates the raw audio format that results from decoding the encoded audio.
- *encoding* specifies the encoding format. This should just be **B_ANY**; this field, and the audio_encoding enum, are depreciated. Instead, the BMediaFormats database should be used to identify the encoding format.
- *bit_rate* specifies the bit rate (not byte rate) of the data stream.
- *frame_size* indicates the size of each frame of audio, in bytes.
- *multi_info* provides multi-channel information.

---

media_encoded_audio_header

Declared in: be/media/MediaDefs.h

```
struct media_encoded_audio_header {
    int32 _reserved_0[14];
    uint32 buffer_flags;
    uchar unused_mask;
    uchar _reserved_2[3];
    };
```

Defines the header of a an encoded audio buffer.

- *buffer_flags* contains flags that provide additional information about the buffer. The only value currently defined (other than 0) is **B_MEDIA_KEY_FRAME**, which indicates that the buffer is a key frame.
- *unused_mask* indicates which bits of the last byte in the buffer are unused (since encoded audio may result in a number of bits that doesn't fill up the last byte in the buffer).

If you wish, you can add more fields to the end of the header.

---

media_encoded_video_format

Declared in: be/media/MediaDefs.h

```
struct media_encoded_video_format {
    enum video_encoding {
      B_ANY
    };
    media_raw_video_format output;
    float avg_bit_rate;
    float max_bit_rate;
    video_encoding encoding;
    size_t frame_size;
    int16 forward_history;
    int16 backward_history;
    uint32 _reserved_[3];
    static media_encoded_video_format wildcard;
    };
```

Describes the format of encoded video data.

- *output* specifies the raw video format that results from decoding the video. Set this to **media_raw_video_format::wildcard**.
- *avg_bit_rate* indicates the average bit rate of the video, and *max_bit_rate* specifies the maximum bit rate that occurs at any point in the video.

- *encoding* has been depreciated, and should always be **B_ANY**; use the BMediaFormats lookup database to identify the encoding format.
- *frame_size* indicates the size in bytes of a frame of data.
- *forward_history* indicates the maximum number of future frames the encoding algorithm requires in order to decode the current frame.
- *backward_history* indicates the maximum number of past frames the encoding algorithm requires in order to decode the current frame.

---

media_encoded_video_header

Declared in: be/media/MediaDefs.h

```
struct media_encoded_video_header {
    int32 _reserved_1[9];
    uint32 field_flags;
    int16 forward_history;
    int16 backward_history;
    uchar unsued_mask;
    uchar _rserved_2[3];
    float field_gamma;
    uint32 field_sequence;
    uint16 field_number;
    uint16 pulldown_number;
    uint16 first_active_line;
    uint16 line_count;
    };
```

Defines the header of a video buffer. The fields in this buffer are the same as those in the **media_video_header** structure, except for the reserved fields, the *forward_history* and *backward_history* fields, the *unused_mask*, which indicates which bits are unused in the last byte of the video buffer, and *field_flags*, which can be either 0 or **B_MEDIA_KEY_FRAME** (which indicates that the field is a key frame).

The *forward_history* field indicates how many buffers ahead of the current one are required in order to decode the current buffer. The *backward_history* field indicates the number of previous buffers are required to decode the current buffer.

- *field_gamma* specifies the gamma at which the video was captured.
- *field_sequence* is an identifying number assigned to the field when it was captured. They progress in sequence; you can tell if a field was dropped by looking for skips in the sequence. Note that this value doesn't necessarily start at 0, and it can wrap back around to 0.
- *field_number* indicates which field of a video frame the buffer represents. It's 0 for field 1, 1 for field 2, and so forth.

- *pulldown_number* indicates that the field represented by the buffer is a duplicate of a field pulldown_number fields back. If this value is 0, it's not a duplicate field. This is used in cases when the video has been converted from one field rate to another, resulting in duplication of fields; these duplicated fields can be marked using pulldown numbers so video hardware and software can optimize their performance by knowing that duplicate fields exist.

- *first_active_line* indicates the topmost line that actually contains video data (instead of VBL data); this is an NTSC or PAL line number, which is 1-based. *line_count* indicates the number of active video lines in the field.

You may add fields to the end of this buffer if your video format requires them.

---

media_file_format

Declared in: be/media/MediaDefs.h

```
struct media_file_format {
    enum {
      B_READABLE = 0x1,
      B_WRITABLE = 0x2,
      B_PERFECTLY_SEEKABLE = 0x4,
      B_IMPERFECTLY_SEEKABLE = 0x8,
      B_KNOWS_RAW_VIDEO = 0x10,
      B_KNOWS_RAW_AUDIO = 0x20,
      B_KNOWS_MIDI = 0x40,
      B_KNOWS_ENCODED_VIDEO = 0x80,
      B_KNOWS_ENCODED_AUDIO = 0x100,
      B_KNOWS_OTHER = 0x1000000,
      B_KNOWS_ANYTHING = 0x2000000
    };
    uint32 capabilities;
    media_file_format_id id;
    media_format_family family;
    int32 version;

    uint32 _reserved_[25];
    char mime_type[64];
    char pretty_name[64];
    char short_name[32];
    char file_extension[8];
    char reserved[88];
    };
```

Describes a media file format.

- *capabilities* is a bitfield providing informational flags describing the file format's capabilities.

- **B_READABLE** indicates that the file format is readable.
- **B_WRITABLE** indicates that the format can be written.
- **B_PERFECTLY_SEEKABLE** indicates that seek operations are precise.
- **B_IMPERFECTLY_SEEKABLE** indicates that seek operations may not be totally accurate.
- **B_KNOWS_RAW_VIDEO** indicates that the file format can handle video.
- **B_KNOWS_RAW_AUDIO** indicates that the format can handle audio.
- **B_KNOWS_MIDI** indicates that the format can handle MIDI data.
- **B_KNOWS_ENCODED_VIDEO** indicates that the file format can handle encoded video.
- **B_KNOWS_ENCODED_AUDIO** indicates that the file format can handle encoded audio.
- **B_KNOWS_OTHER** indicates that the format can handle other media information, such as text, clipping, control changes, and so forth.
- **B_KNOWS_ANYTHING** indicates that the format can handle any kind of media data.
- *id* is an opaque ID used by the Media Kit to construct a BMediaFile object.
- *family* is one of the **media_format_family** constants.
- *version* should be 100.
- *mime_type* specifies the MIME type string that identifies the file format.
- *pretty_name* is a nice user-viewable name, such as "QuickTime File Format".
- *short_name* is a short-form name, like "QuickTime".
- *file_extension* specifies the file extension used on files containing this format, such as ".mov".

---

media_file_format_id

Declared in: be/media/MediaDefs.h

```
struct media_file_format_id {
    ino_t node;
    dev_t device;
    uint32 internal_id;
    };
```

A simple means for identifying file formats; used by the **media_file_format** structure.

---

media_format

Declared in: be/media/MediaDefs.h

```
struct media_format {
    media_type type;
    type_code user_data_type;
    uchar user_data[48];
    uint32 _reserved_[3];
    uint16 require_flags;
    uint16 deny_flags;

    private:
    void *meta_data;
    int32 meta_data_size;
    area_id meta_data_area;
    area_id use_area;
    team_id team;
    void *thisPtr;

    public:
    union {
      media_multi_audio_format raw_audio;
      media_raw_video_format raw_video;
      media_multistream_format multistream;
      media_encoded_audio_format encoded_audio;
      media_encoded_video_format encoded_video;
      char _reserved_[96];
    } u;
    bool IsVideo() const ;
    uint32 Width() const;
    uint32 Height() const;
    color_space ColorSpace() const;
    uint32 &Width();
    uint32 &Height();
    color_space &ColorSpace();

    bool IsAudio() const;
    uint32 &AudioFormat();
    uint32 AudioFormat() const;
    uint32 Encoding() const;
    bool Matches(const media_format *otherFormat) const;
    void SpecializeTo(const media_format *otherFormat);
    void SetMetaData(const void *data, int32 size);
    const void *MetaData() const;
    int32 MetaDataSize() const;

    media_format();
    media_format(const media_format &other);
    ~media_format();
```

```
media_format & operator=(const media_format &clone);
};
```

The media_format structure can describe any media data format the BeOS can support.

- *type* indicates the type of media the format describes.
- *user_data_type* and *user_data* allow the application to store custom data in the media_format structure; user_data_type specifies the type of data stored in user_data (**B_INT32_TYPE**, **B_STRING_TYPE**, etc).
- *require_flags* and *deny_flags* are media_format_flags indicating what options are required, and what options are rejected.
- *meta_data*, *meta_data_size*, *meta_data_area*, *use_area*, *team*, and *thisPtr* are currently only used by decoders when communicating with the file reader; they're not currently for public use.
- The union, *u*, permits the selection of a specific media format, such as raw audio or multistream data, and contains the descriptive information specific to that media format.

If any field is 0, it's treated as a wildcard.

There are several convenience functions for fetching information from the format that take care of dealing with different types automatically:

- `IsVideo()` returns true if the format represents video (either raw or encoded), otherwise it returns false.
- `Width()` returns the width of the video data; its value is undetermined if the format doesn't represent video.
- `Height()` returns the height of the video data; its value is undetermined if the format doesn't represent video.
- `ColorSpace()` returns the video's color space. Its value is undetermined if the format doesn't represent video.
- `IsAudio()` returns true if the format represents audio (either raw or encoded), otherwise it returns false.
- `AudioFormat()` returns the audio's sample format.
- `Matches()` returns **`true`** if the format matches the specified *otherFormat*, otherwise it returns **`false`**.
- `SpecializeTo()` adjusts the specified *otherFormat* as needed to match the format.

media_header

Declared in: be/media/MediaDefs.h

```
struct media_header {
    media_type type;
    media_buffer_id buffer;
    int32 destination;
    media_node_id time_source;
    uint32 _depreciated_
        uint32 size_used;
    bigtime_t start_time;
    area_id owner;
    enum {
      B_SEEK_TAG = 'TRST'
    };
    type_code user_data_type;
    uchar user_data[64];
    uint32 _reserved_[2];
    off_t file_pos;
    size_t orig_size;
    uint32 data_offset;
    union {
      media_audio_header raw_audio;
      media_video_header raw_video;
      media_multistream_header multistream;
      media_encoded_audio_header encoded_audio;
      media_encoded_video_header encoded_video;
      char _reserved_[64];
    } u;
    };
```

Defines the header of a media data buffer.

- *type* indicates the type of media the format describes.
- *buffer* indicates what buffer the header goes with. This field is filled out by the `BBufferProducer::SendBuffer()` function.
- *destination* specifies the ID number of the media_destination to which the buffer is being sent. This field is filled out by the `BBufferProducer::SendBuffer()` function.
- *time_source* indicates the ID number of the node that encoded the *start_time*.
- *size_used* specifies how many bytes of the buffer actually contain data.
- *start_time* indicates the performance time at which the buffer should be performed.
- *owner* specifies the area ID of the buffer's owner.
- *user_data_type* and *user_data* allow the application to store custom data in the media_format structure; *user_data_type* specifies the type of data stored in user_data

(B_INT32_TYPE, B_STRING_TYPE, etc). The **B_SEEK_TAG** constant indicates that the *user_data* contains a seek tag.

- *file_pos* indicates where in the corresponding file the data came from, and *orig_size* indicates how big it was; if the data didn't come from a file, these should be zero.
- *data_offset* indicates the offset within the buffer to the data. This is reflected by the BBuffer::Data() function.
- The union, *u*, permits the selection of a specific media format, such as raw audio or multistream data, and contains the descriptive information specific to that media format.

---

media_input

Declared in: be/media/MediaDefs.h

```
struct media_input {
    media_input();
    ~media_input();
    media_node node;
    media_source source;
    media_destination destination;
    media_format format;
    char name[B_MEDIA_NAME_LENGTH];
    private:
    _reserved_media_input_[4];
    };
```

The media_input structure describes a complete connection between a media_source and media_destination, from the point-of-view of the consumer (the destination).

- *node* is the media_node that owns the input.
- *source* is the source socket from which the data is being sent.
- *destination* is the destination socket at which the data is arriving.
- *format* is the format of the data.
- *name* is the input's name.

---

media_multistream_format

Declared in: be/media/MediaDefs.h

```
struct media_multistream_format {
    enum {
      B_ANY = 0,
      B_VID = 1,
      B_AVI,
```

```
    B_MPEG1,
    B_MPEG2,
    B_QUICKTIME,
    B_PRIVATE = 90000,
    B_FIRST_USER_TYPE = 100000
  };
  float avg_bit_rate;
  float max_bit_rate;
  uint32 avg_chunk_size;
  uint32 max_chunk_size;
  enum {
    B_HEADER_HAS_FLAGS = 0x1,
    B_CLEAN_BUFFERS = 0x2,
    B_HOMOGENOUS_BUFFERS = 0x4
  };
  uint32 flags;
  int32 format;
  uint32 _reserved_[2];
  struct vid_info {
    float frame_rate;
    uint16 width;
    uint16 height;
    color_space space;
    float sampling_rate;
    uint32 sample_format;
    uint16 byte_order;
    uint16 channel_count;
  };
  struct avi_info {
    uint32 us_per_frame;
    uint16 width;
    uint16 height;
    uint16 _reserved_;
    uint16 type_count;
    media_type types[5];
  };
  union {
    vid_info vid;
    avi_info avi;
  } u;
  static media_multistream_format wildcard;
  };
```

Describes the format of multistream media data, such as a QuickTime, AVI, or MPEG movie.

- *avg_bit_rate* and *max_bit_rate* indicate the average and maximum bit rates of the media data.

- *avg_chunk_size* and *max_chunk_size* indicate the average and maximum buffer sizes; if all buffers are the same size, these two values should be equal.
- *flags* provides informational flags for the buffers. **B_HEADER_HAS_FLAGS** indicates that the buffers' headers have valid flags; **B_CLEAN_BUFFERS** indicates that each buffer represents an integral number of frames, and **B_HOMOGENOUS_BUFFERS** indicates that each buffer has only one format in it.
- *format* indicates the multistream data format (**B_AVI**, **B_QUICKTIME**, etc).
- The union *u* contains either a **vid_info** or an **avi_info** structure that describes the video stream:

vid_info

- *frame_rate* specifies the video frame rate (in frames per second).
- *width* and *height* indicate the width and height of the video, in pixels.
- *space* indicates the video's color space.
- *sampling_rate*, *sample_format*, *byte_order*, and *channel_count* specify the sampling rate, sample format, byte order, and number of channels (stereo or mono) of the media's audio.

avi_info

- *us_per_frame* specifies the number of microseconds each frame should be displayed.
- *width* and *height* indicate the width and height of the video in pixels.
- *type_count* indicates the number of media_types used in the AVI movie.
- *types* is a list of the media_types used.

---

media_multi_audio_format

Declared in: be/media/MediaDefs.h

```
struct media_multi_audio_format {
    media_raw_audio_format raw;
    media_multi_audio_info multi;
    };
```

Describes a multi-channel audio format. This is constructed by using a media_raw_audio_format to describe the physical format of the sound, followed by a media_multi_audio_info structure that adds the multi-channel information. This allows compatibility with existing BeOS software while adding support for multi-channel audio in new software.

---

media_multi_audio_info

Declared in: be/media/MediaDefs.h

```
struct media_multi_audio_info {
    uint32 channel_mask;
    int16 valid_bits;
    uint16 matrix_mask;
    uint32 _reserved_b[3];
    };
```

Provides information about multi-channel audio support.

- *channel_mask* indicates which channels are supported; for each channel (**B_CHANNEL_LEFT**, etc) the corresponding bit is set in this mask.
- *valid_bits* indicates how many bits in a 32-bit integer sample actually contain sample data.
- *matrix_mask* indicates which multi-channel audio matrices are used.

---

media_multistream_header

Declared in: be/media/MediaDefs.h

```
struct media_multistream_header {
    uint32 _reserved_[14];
    uchar unused_mask;
    uchar _reserved_2[3];
    enum {
      B_MASTER_HEADER = 0x1,
      B_SUBSTREAM_HEADER = 0x2,
      B_COMPLETE_BUFFER = 0x4
    };
    uint32 flags;
    };
```

Defines the header of a multistream media buffer.

- *unused_mask* indicates which bits of the last byte in the buffer are unused.
- *flags* provides informational flags about the buffer. If **B_MASTER_HEADER** is specified, master stream header data is in the buffer; **B_SUBSTREAM_HEADER** indicates that the buffer contains substream header information. **B_COMPLETE_BUFFER** indicates that the buffer's data represents an integral number of frames.

You may add fields to the end of this structure if your multistream media format requires them.

---

media_node

Declared in: be/media/MediaDefs.h

```
class media_node {
    public:
    media_node();
    ~media_node();
    media_node_id node;
    port_id port;
    uint32 kind;
    static media_node null;
    private:
    uint32 _reserved_[3];
    };
```

The `media_node` structure is the class that defines a node to client applications; interactions with the BMediaRoster regarding nodes is done using `media_node` objects.

- *node* indicates the node's ID number.
- *port* specifies the port ID number of the node's control port.
- *kind* specifies the node's media kind. See "node_kind" on page182 for a list of node kinds.
- *null* represents a null node

Applications will usually treat this as a black box.

---

media_node_attribute

Declared in: be/media/MediaNode.h

```
struct media_node_attribute {
      enum {
        B_R40_COMPILED = 1,
        B_USER_ATTRIBUTE_NAME = 0x1000000,
        B_FIRST_USER_ATTRIBUTE
          };
      uint32 what;
      uint32 flags;
      uint64 data;
    };
```

The media_node_attribute structure provides additional information about nodes. Nodes can have as many of these as they want, and will return them from their `BMediaNode::GetNodeAttributes()` function. Applications query the node using the `BMediaRoster::GetNodeAttributesFor()` function.

The what field indicates the attribute type. `B_R40_COMPILED` indicates that the node was compiled using BeOS Release 4.0; BMediaNode returns this attribute automatically for nodes

that were compiled using the Release 4.0 headers. Nodes may create their own attribute types, beginning at **B_FIRST_USER_ATTRIBUTE**.

The **B_USER_ATTRIBUTE_NAME** attribute specifies the name of a protocol for the user attributes implemented by the node. For example, if you devise a third-party standard for attribute values whose attribute IDs are greater than **B_FIRST_USER_ATTRIBUTE**, the value of **B_USER_ATTRIBUTE_NAME** would be the name of the standard.

The flags and data fields are used for different things depending on the what value; the **B_R40_COMPILED** attribute doesn't use them.

---

media_node_id

Declared in: be/media/MediaDefs.h

```
typdef int32 media_node_id;
```

A node ID number.

---

media_output

Declared in: be/media/MediaDefs.h

```
struct media_output {
    media_output();
    ~media_output();
    media_node node;
    media_source source;
    media_destination destination;
    media_format format;
    char name[B_MEDIA_NAME_LENGTH];

    private:
    _reserved_media_output_[4];
    };
```

The media_output structure describes a complete connection between a media_source and media_destination, from the point-of-view of the consumer (the destination).

- *node* is the media_node that owns the output.
- *source* is the source socket from which the data is being sent.
- *destination* is the destination socket to which the data is being sent.
- *format* is the format of the data.
- *name* is the output's name.

---

media_raw_audio_format

Declared in: be/media/MediaDefs.h

```
struct media_raw_audio_format {
    enum {
      B_AUDIO_FLOAT = 0x24,
      B_AUDIO_INT = 0x4
      B_AUDIO_SHORT = 0x2,
      B_AUDIO_UCHAR = 0x11,
      B_AUDIO_CHAR = 0x1,
    B_AUDIO_SIZE_MASK = 0xf
    }
    float frame_rate;
    uint32 channel_count;
    uint32 format;
    uint32 byte_order;
    size_t buffer_size;

    static media_multi_audio_format wildcard;
    };
```

Describes the format of raw audio data.

The *frame_rate* indicates the frame rate, in Hertz. One frame is one sample per channel (stereo buffers contain two samples per channel). For audio, each buffer contains multiple audio samples, whereas video contains one frame, or one field (half-frame) per buffer.

The *channel_count* indicates the number of channels of audio (typically this is either 1 for mono or 2 for stereo).

The *format* specifies the sample format (for compressed formats, see media_encoded_audio_format):

- **B_AUDIO_FLOAT**. Each sample is four bytes; 0 is the middle, -1.0 is the bottom, 1.0 is the top.
- **B_AUDIO_INT**. Each sample is four bytes; 0 is the middle, 0x80000001 is the bottom, 0x7FFFFFFF is the top.
- **B_AUDIO_SHORT**. Each sample is two bytes; 0 is the middle, -32767 is the bottom, 32767 is the top.
- **B_AUDIO_UCHAR**. Each sample is one byte; 128 is the midpoint, 1 is the bottom, 255 is the top.
- **B_AUDIO_CHAR**. Each sample is one byte; 0 is the midpoint, -127 is the bottom, 127 is the top.

It's guaranteed that (*format* & **B_AUDIO_SIZE_MASK**) will always be equal to the size of the sample.

The *byte_order* indicates the endianness of the data (either **B_MEDIA_BIG_ENDIAN** or **B_MEDIA_LITTLE_ENDIAN**).

buffer_size indicates the size of each buffer, in bytes.

---

media_raw_video_format

Declared in: be/media/MediaDefs.h

```
struct media_raw_video_format {
    float field_rate;
    uint32 interlace;
    uint32 first_active;
    uint32 last_active;
    uint32 orientation;
    uint16 pixel_width_aspect;
    uint16 pixel_height_aspect;
    media_video_display_info display

    static media_raw_video_format wildcard;
    };
```

Describes the format of raw video data.

- *field_rate* indicates the field rate. This is 59.94 for NTSC and 50.0 for PAL. If you use BT848 in non-interlaced mode (thereby picking out only every other field), you'd specify 29.97.
- *interlace* specifies how many fields there are per frame. A value of 1 indicates that the video is progressive (non-interlaced). NTSC video is interlaced such that there are two fields per frame, so for standard NTSC video, this value is 2.
- *first_active* and *last_active* indicate where the VBL lines at the top of the frame end, and where they resume at the bottom of the frame; video data is confined between these two scan lines. Usually you'll use 0 for *first_active* and *line_count*-1 for *last_active*; these indicate wildcard or "don't care" values.
- *orientation* indicates the orientation of the video; this is typically **B_VIDEO_TOP_LEFT_RIGHT**.
- *pixel_width_aspect* and *pixel_height_aspect* specify the numerator and denominator of the video's aspect ratio. If the aspect ratio is 4:3, these values will be 4 and 3, respectively.

---

media_request_info

Declared in: be/media/MediaNode.h

```
struct media_request_info {
    enum what_code {
      B_SET_VIDEO_CLIPPING_FOR = 1,
      B_REQUEST_FORMAT_CHANGE,
```

```
    B_SET_OUTPUT_ENABLED,
    B_SET_OUTPUT_BUFFERS_FOR,
    B_FORMAT_CHANGED = 4097
  };
  what_code what;
  int32 change_tag;
  status_t status;
  int32 cookie;
  void * user_data;
  media_source source;
  media_destination destination;
  media_format format;
  uint32 _reserved_[32];
  };
```

This structure contains information that describes a request that has been completed (or has failed). It gets passed into the `BMediaNode::RequestCompleted()` function.

The `what` field describes what type of request has been completed:

- **`B_SET_VIDEO_CLIPPING_FOR`** indicates that a `BBufferConsumer::SetVideoClippingFor()` request has been completed. The `source` and `destination` fields contain these values from the request, and `format.u.raw_video.display` describes the new video display settings.
- **`B_REQUEST_FORMAT_CHANGE`** indicates that a **`BBufferConsumer::RequestFormatChange()`** request has been completed. The `source` and `destination` fields contain these values from the request, and `format` contains the negotiated format (it may have been altered by the producer if there were wildcard fields).
- **`B_SET_OUTPUT_ENABLED`** indicates that a `BBufferConsumer::SetOutputEnabled()` request has been completed. The `source` and `destination` fields contain these values from the request, and `status` is always `B_OK`.
- **`B_SET_OUTPUT_BUFFERS_FOR`** indicates that a `BBufferConsumer::SetOutputBuffersFor()` request has been completed. The `source` and `destination` fields contain these values from the request, and `cookie` is a pointer to the BBufferGroup that was set. Note that the change was only applied if `status` is `B_OK`.
- **`B_FORMAT_CHANGED`** indicates that a `BBufferConsumer::FormatChanged()` request has been completed. The `source` and `destination` fields contain these values from the request, and `format` indicates the new format. The values seen here are from the initator's point of view.

The `user_data` field contains the *userData* pointer passed to the function that initiated the request; this can be anything the node making the request wants.

`status` indicates the status of the function called, and is `B_OK` if no error occurred.

---

media_seek_tag

Declared in: be/media/MediaDefs.h

```
struct media_seek_tag {
    char data[16];
    };
```

The `media_seek_tag` type is used to contain the seek tag data placed in buffer headers by producers. This can contain whatever information the producer deems necessary.

---

media_source

Declared in: be/media/MediaDefs.h

```
struct media_source {
    media_source(port_id, int32);
    media_source(const media_source &clone);
    media_source & operator=  (const media_source &clone);
    media_source();
    ~media_source();
    port_id port;
    int32 id;
    static media_source null;
    private:
    uint32 _reserved_media_source_[2];
    };
```

Describes the source end of a media connection between two nodes. The `port` field indicates the port ID on which the source communicates. This must be the same as the media_node.port value.

---

media_video_display_info

Declared in: be/media/MediaDefs.h

```
struct media_video_display_info {
    color_space format;
    uint32 line_width;
    uint32 line_count;
    uint32 bytes_per_row;
    uint32 pixel_offset;
    uint32 line_offset;
    uint32 flags;
    uint32 _reserved_[3];
    static media_video_display_info wildcard;
    };
```

Describes a video buffer in terms of how its contents should be interpreted in order to display it on the screen.

- *format* specifies the buffer's color space.
- *line_width* and *line_count* indicate the total width of the buffer in pixels, and the total height of the buffer in lines.
- *bytes_per_row* specifies the number of bytes wide the buffer's image is, including any slop space.
- *pixel_offset* indicates the number of pixels between the start of the buffer and the start of the video field. The buffer is the framebuffer.
- *line_offset* specifies the offset between the start of the buffer and the start of the field, in lines. The buffer is the framebuffer.
- *flags* are media_display_flags, indicating the dominance and order of the video fields.

---

media_video_header

Declared in: be/media/MediaDefs.h

```
struct media_video_header {
    uint32 _reserved_[12];
    float field_gamma;
    uint32 field_sequence;
    uint16 field_number;
    uint16 pulldown_number;
    uint16 first_active_line;
    uint16 line_count;
    };
```

Defines the header of a video buffer.

- *field_gamma* specifies the gamma at which the video was captured.
- *field_sequence* is an identifying number assigned to the field when it was captured. They progress in sequence; you can tell if a field was dropped by looking for skips in the sequence. Note that this value doesn't necessarily start at 0, and it can wrap back around to 0.
- *field_number* indicates which field of a video frame the buffer represents. It's 0 for field 1, 1 for field 2, and so forth.
- *pulldown_number* indicates that the field represented by the buffer is a duplicate of a field pulldown_number fields back. If this value is 0, it's not a duplicate field. This is used in cases when the video has been converted from one field rate to another, resulting in duplication of fields; these duplicated fields can be marked using pulldown numbers so video hardware and software can optimize their performance by knowing that duplicate fields exist.

- *first_active_line* indicates the topmost line that actually contains video data (instead of VBL data); this is an NTSC or PAL line number, which is 1-based. *line_count* indicates the number of active video lines in the field.

You may add fields to the end of this buffer if your video format requires them.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Ports

Declared in: be/kernel/OS.h
Library: libroot.so
Summary: *more...*

A port is a system-wide message repository into which any thread can copy a buffer of data, and from which any thread can then retrieve the buffer. This repository is implemented as a first-in/first-out message queue: A port stores its messages in the order in which they're received, and it relinquishes them in the order in which they're stored. Each port has its own message queue.

Ports are largely subsumed by the Application Kit's BMessage class (and relatives). The two features of ports that you can't get at the BMessage level are:

- Ports let you set the length of the message queue.
- Ports can be used in C code (as opposed to C++).

For most applications, these are inessential additions.

For more information on ports, see "Port Concepts".

Port Functions

create_port()

port_id **create_port(**int32 *queue_length*, const char **name*)

Creates a new port and returns its **port_id** number. The port's name is set to *name* and the length of its message queue is set to *queue_length*. Neither the name nor the queue length can be changed once they're set. The name shouldn't exceed B_OS_NAME_LENGTH (32) characters.

In setting the length of a port's message queue, you're telling it how many messages it can hold at a time. When the queue is filled—when it's holding *queue_length* messages—subsequent invocations of write_port() (on that port) block until room is made in the queue (through calls to read_port()) for the additional messages. Once the queue length is set by `create_port()`, it can't be changed.

This function sets the owner of the port to be the team of the calling thread. Ownership can subsequently be transferred through the set_port_owner() function. When a port's owner dies (when all the threads in the team are dead), the port is automatically deleted. If you want to delete a port prior to its owner's death, use the delete_port() function.

RETURN CODES

- B_BAD_VALUE. *queue_length* is too big or less than zero.
- B_NO_MORE_PORTS. The system couldn't allocate another port.

close_port()

status_t **close_port(**port_id *port*)

Closes the *port* so no more messages can be written to it. After you close a port, you can call read_port() on it, but a write_port() call will return B_BAD_PORT_ID. You can't reopen a closed port; you call this function so you can read through a port's unread messages prior to deleting the port, while ensuring that no new messages will show up. After you've read the messages, you should call delete_port() on *port*.

RETURN CODES

- B_OK. *port* is now closed.
- B_BAD_PORT_ID. *port* doesn't identify an open port.

delete_port()

status_t **delete_port(port_id** *port***)**

Deletes the given port. The port's message queue doesn't have to be empty—you can delete a port that's holding unread messages. Threads that are blocked in `read_port()` or `write_port()` calls on the port are automatically unblocked (and return `B_BAD_SEM_ID`).

The thread that calls `delete_port()` doesn't have to be a member of the team that owns the port; any thread can delete any port.

RETURN CODES

- **`B_OK`.** The port was deleted.
- `B_BAD_PORT_ID`. *port* isn't a valid port.

---

find_port()

port_id **find_port(**const char **port_name***)*

Returns the **port_id** of the named port. port_name should be no longer than 32 characters (`B_OS_NAME_LENGTH`).

RETURN CODES

- `B_NAME_NOT_FOUND`. *port_name* doesn't name an existing port.

---

get_port_info() , get_next_port_info()

status_t **get_port_info(port_id** *port*, **port_info** **info***)*

status_t **get_next_port_info(team_id** *team*,
uint32 **cookie*,
port_info **info***)*

Copies information about a particular port into the `port_info` structure designated by *info*. The first version of the function designates the port directly, by **port_id**.

The `get_next_port_info()` version lets you step through the list of a team's ports through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns `B_BAD_VALUE` when there are no more ports to visit:

```
/* Get the port_info for every port in this team. */
port_info info;
int32 cookie = 0;
```

```
while (get_next_port_info(0, &cookie, &info) == B_OK)
    ...
```

The information in the `port_info` structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a port as it exists just before the info-retrieving function returns.

RETURN CODES

- `B_OK`. The port was found; *info* contains valid information.
- `B_BAD_VALUE`. *port* doesn't identify an existing port, *team* doesn't identify an existing team, or there are no more ports to visit.

---

port_buffer_size() , port_buffer_size_etc()

```
ssize_t port_buffer_size(port_id port)
ssize_t port_buffer_size_etc(port_id port,
    uint32 flags,
    bigtime_t timeout)
```

These functions return the length (in bytes) of the message buffer that's at the head of *port*'s message queue. You call this function in order to allocate a sufficiently large buffer in which to retrieve the message data.

The `port_buffer_size()` function blocks if the port is currently empty. It unblocks when a `write_port()` call gives this function a buffer to measure (even if the buffer is 0 bytes long), or when the port is deleted.

The `port_buffer_size_etc()` function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **`B_TIMEOUT`** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

RETURN CODES

- `B_BAD_PORT_ID`. *port* doesn't identify an existing port, or the port was deleted while the function was blocked.
- `B_TIMED_OUT`. The *timeout* limit expired.
- `B_WOULD_BLOCK`. You asked for a *timeout* of 0, but there are no messages in the queue.

See also: `read_port()`

---

port_count()

```
int32 port_count(port_id port)
```

Returns the number of messages that are currently in *port*'s message queue. This is the number of messages that have been written to the port through calls to `write_port()` but that haven't yet been picked up through corresponding `read_port()` calls.

This function is provided mostly as a convenience and a semi-accurate debugging tool. The value that it returns is inherently undependable: There's no guarantee that additional `read_port()` or `write_port()` calls won't change the count as this function is returning.

RETURN CODES

- `B_BAD_PORT_ID`. *port* doesn't identify an existing port.

See also: `get_port_info()`

---

read_port() , read_port_etc()

```
ssize_t read_port(port_id port,
    int32 *msg_code,
    void *msg_buffer,
    size_t buffer_size)
```

```
ssize_t read_port_etc(port_id port,
    int32 *msg_code,
    void *msg_buffer,
    size_t buffer_size,
    uint32 flags,
    bigtime_t timeout)
```

These functions remove the message at the head of *port*'s message queue and copy the messages's contents into the *msg_code* and *msg_buffer* arguments. The size of the *msg_buffer* buffer, in bytes, is given by *buffer_size*. It's up to the caller to ensure that the message buffer is large enough to accommodate the message that's being read. If you want a hint about the message's size, you should call `port_buffer_size()` before calling this function.

If *port*'s message queue is empty when you call **`read_port()`**, the function will block. It returns when some other thread writes a message to the port through `write_port()`. A blocked read is also unblocked if the port is deleted.

The `read_port_etc()` function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **`B_TIMEOUT`** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

RETURN CODES

A successful call returns the number of bytes that were written into the *msg_buffer* argument.

- `B_BAD_PORT_ID`. *port* doesn't identify an existing port, or the port was deleted while the function was blocked.

- `B_TIMED_OUT`. The *timeout* limit expired.
- `B_WOULD_BLOCK`. You asked for a *timeout* of 0, but there are no messages in the queue.

See also: `write_port()`, `port_buffer_size()`

---

set_port_owner()

```
status_t set_port_owner(port_id port, team_id team)
```

Transfers ownership of the designated port to *team*. A port can only be owned by one team at a time; by setting a port's owner, you remove it from its current owner.

There are no restrictions on who can own a port, or on who can transfer ownership. In other words, the thread that calls `set_port_owner()` needn't be part of the team that currently owns the port, nor must you only assign ports to the team that owns the calling thread (although these two are the most likely scenarios).

Port ownership is meaningful for one reason: When a team dies (when all its threads are dead), the ports that are owned by that team are freed. Ownership, otherwise, has no significance—it carries no special privileges or obligations.

To discover a port's owner, use the `get_port_info()` function.

RETURN CODES

- `B_OK`. Ownership was successfully transferred.
- `B_BAD_PORT_ID`. *port* doesn't identify a valid port.
- `B_BAD_TEAM_ID`. *team* doesn't identify a valid team.

See also: `get_port_info()`

---

write_port() , write_port_etc()

```
status_t write_port(port_id port,
    int32 msg_code,
    void *msg_buffer,
    size_t buffer_size)
```

```
status_t write_port_etc(port_id port,
    int32 msg_code,
    void *msg_buffer,
    size_t buffer_size,
    uint32 flags,
    bigtime_t timeout)
```

These functions place a message at the tail of *port*'s message queue. The message consists of *msg_code* and *msg_buffer*:

- *msg_code* holds the "message code." This is a mask, flag, or other predictable value that gives a general representation of the message.
- *msg_buffer* is a pointer to a buffer that can be used to supply additional information. You pass the length of the buffer, in bytes, as the value of the *buffer_size* argument. The buffer can be arbitrarily long.

If the port's queue is full when you call `write_port()`, the function will block. It returns when a `read_port()` call frees a slot in the queue for the new message. A blocked `write_port()` will also return if the target port is deleted or closed.

The `write_port_etc()` function lets you set a limit on the amount of time the function will wait for a free queue slot. To set the limit, you pass **`B_TIMEOUT`** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

RETURN CODES

- `B_OK`. The port was successully written to.
- `B_BAD_PORT_ID`. *port* doesn't identify an open port, or the port was deleted while the function was blocked.
- `B_TIMED_OUT`. The *timeout* limit expired.
- `B_WOULD_BLOCK`. You asked for a *timeout* of 0, but there are no free slots in the message queue.

See also: `read_port()`

Port Structures and Constants

port_info

```
struct {
    port_id port;
    team_id team;
    char name[B_OS_NAME_LENGTH];
    int32 capacity;
    int32 queue_count;
    int32 total_count;
    } port_info
```

The port_info structure provides information about a port. You retrieve one of these structures through `get_port_info()` or `get_next_port_info()`.

- **`port`** . The **`port_id`** number of the port.
- **`team`** . The `team_id` of the port's owner.
- **`name`** . The name assigned to the port.
- **`capacity`** . The length of the port's message queue.
- **`queue_count`** . The number of messages currently in the queue.
- **`total_count`** . The total number of message that have been read from the port.

Note that the **`total_count`** number doesn't include the messages that are currently in the queue.

The Be Book The Kernel Kit The Kernel Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Ports Function Summary

Function Summary

Function	Description
close_port()	Closes a port so no more messages can be written to it
create_port()	Creates a new port
delete_port()	Deletes a port
find_port()	Finds a port given its name
get_next_port_info()	Enumerates information about all ports in a given team
get_port_info()	Returns information about a specific port
port_buffer_size()	Returns the size of the message at the head of the queue (blocks)
port_buffer_size_etc()	Returns the size of the message at the head of the queue (blocks with timeout)
port_count()	Returns the number of messages currently in the queue
read_port()	Reads and removes the message at the head of the queue (blocks)
read_port_etc()	Reads and removes the message at the head of the queue (blocks with timeout)
set_port_owner()	Transfers ownership of a port
write_port()	Places a message at the tail of a port (blocks)

write_port_etc()	Places a message at the tail of a port (blocks with timeout)

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Port Concepts

A port is a system-wide message repository into which any thread can copy a buffer of data, and from which any thread can then retrieve the buffer. This repository is implemented as a first-in/first-out message queue: A port stores its messages in the order in which they're received, and it relinquishes them in the order in which they're stored. Each port has its own message queue.

Creating and Destroying a Port

The `create_port()` function creates a new port and assigns it a unique, system-wide **`port_id`** number. Although ports are accessible to all threads, the **`port_id`** numbers aren't disseminated by the operating system—there's no "find port" function. If you create a port and want some other thread to be able to write to or read from it, you have to broadcast the **`port_id`** number to that thread.

A port is owned by the team in which it was created. When a team dies (when all its threads are killed), the ports that belong to the team are deleted. A team can bestow ownership of its ports to some other team through the `set_port_owner()` function.

If you want explicitly get rid of a port, you call `delete_port()`. You can delete any port, not just those that are owned by the team of the calling thread. When you delete a port, all of its unread messages are thrown away. If you want to read this messages, but you don't want any new messages to arrive in the meantime, you should call `close_port()` before deleting the port. Note that you can't reopen a closed port; after you get done reading the port's messages, you're expected to delete the port.

The Message Queue: Reading and Writing Port Messages

The length of a port's message queue—the number of messages that it can hold at a time—is set when the port is created.

The functions `write_port()` and `read_port()` manipulate a port's message queue: `write_port()` places a message at the tail of the port's message queue; `read_port()` removes the message at the head of the queue and returns it the caller.`write_port()` blocks if the queue is full; it returns when room is made in the queue by an invocation of `read_port()`. Similarly, if the queue is empty, `read_port()` blocks until `write_port()` is called.

You can provide a timeout for your port-writing and port-reading operations by using the "full-blown" functions `write_port_etc()` and `read_port_etc()`. By supplying a timeout, you can ensure that your port operations won't block forever.

Although each port has its own message queue, all ports share a global "queue slot" pool—there are only so many message queue slots that can be used by all ports taken cumulatively. If too many port queues are allowed to fill up, the slot pool will drain, which will cause `write_port()` calls on less-than-full ports to block. To avoid this situation, you should make sure that your `write_port()` and `read_port()` calls are reasonably balanced.

The write_port() and read_port() functions are the only way to traverse a port's message queue. There's no notion of "peeking" at the queue's unread messages, or of erasing messages that are in the queue.

Port Messages

A port message—the data that's sent through a port—consists of a "message code" and a "message buffer." Either of these elements can be used however you like, but they're intended to fit these purposes:

- The message code (a single four-byte value) should be a mask, flag, or other predictable value that gives a general representation of the flavor or import of the message. For this to work, the sender and receiver of the message must agree on the meanings of the values that the code can take.
- The data in the message buffer can elaborate upon the code, identify the sender of the message, or otherwise supply additional information. The length of the buffer isn't restricted. To get the length of the message buffer that's at the head of a port's queue, you call the port_buffer_size() function.

The message that you pass to write_port() is copied into the port. After write_port() returns, you may free the message data without affecting the copy that the port holds.

When you read a port, you have to supply a buffer into which the port mechanism can copy the message. If the buffer that you supply isn't large enough to accommodate the message, the unread portion will be lost—the next call to read_port() won't finish reading the message.

You typically allocate the buffer that you pass to read_port() by first calling port_buffer_size(), as shown below:

```
char *buf = NULL;
ssize_t size;
int32 code;

/* We'll assume that my_port is valid.
* port_buffer_size() will block until a message shows up.
*/
if ((size = port_buffer_size(my_port)) < B_OK)
   /* Handle the error */

if (size > 0)
   buf = (char *)malloc(size);

if (buf) {
   /* Now we can read the buffer. */
   if (read_port(my_port, &code, (void *)buf, size) < B_OK)
   /* Handle the error */
```

Obviously, there's a race condition (in the example) between `port_buffer_size()` and the subsequent `read_port()` call—some other thread could read the port in the interim. If you're going to use `port_buffer_size()` as shown in the example, you shouldn't have more than one thread reading the port at a time.

As stated in the example, `port_buffer_size()` blocks until a message shows up. If you don't want to (potentially) block forever, you should use the `port_buffer_size_etc()` version of the function. As with the other `...etc()` functions, `port_buffer_size_etc()` provides a timeout option.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Miscellaneous Functions and Constants

Declared in: be/kernel/OS.hheaders/be/kernel/OS.h
Summary: *more...*

Functions

clear_caches()

Declared in: be/kernel/image.h

```
void clear_caches(void *addr, size_t len, uint32 flags)
```

This function clears or invalidates the instruction and data caches. You should only need this function if you're generating code on the fly, or if you're performing a timing loop and you want to start with fresh caches (to get a "worst case" estimate).

The argument are:

- *addr* is the starting address of a section of memory that corresponds to a section of one of the caches.
- *len* is the length, in bytes, of the instruction or data segment that you want to clear or invalidate.
- *flags* is one or both of **B_INVALIDATE_ICACHE** and **B_FLUSH_DCACHE**.

By invalidating a section of the instruction cache, you cause the instructions in that section to be reloaded next time they're needed. Flushing the data cache causes the in-memory copy of the data to be written out to the cache.

debugger()

```
void debugger(const char *string)
```

Throws the calling thread into the debugger. The *string* argument becomes the debugger's first utterance.

disable_debugger()

```
int disable_debugger(int state)
```

Instructs the kernel to send a signal for all exceptions, even those that don't normally trigger the debugger. If the application doesn't have a handler installed for the exception, the team dies without triggering the debugger. *state* should be nonzero to turn on this functionality or 0 to turn it off.

set_alarm()

bigtime_t **set_alarm(**bigtime_t *time*, uint32 *mode***)**

Tells the kernel to send the **SIGALRM** signal at some point in the future, as defined by the arguments:

- If *mode* is **B_PERIODIC_ALARM**, the signal is sent every *time* microseconds, starting as soon as **set_alarm()** function returns.
- If *mode* is **B_ONE_SHOT_ABOLUTE_ALARM**, the signal is sent once (only) after *time* microseconds have elapsed measured from the time the system was booted. If that point has already passed, the signal is sent immediately.
- If *mode* is **B_ONE_SHOT_RELATIVE_ALARM**, the signal is sent once (only) after *time* microseconds have elapsed from the time **set_alarm()** returns.

When the signal is sent, the **SIGALRM** handler is called (you set the handler through the normal means, by calling the Posix **signal()** function). The handler runs in the thread that set the alarm.

From within the **SIGALRM** handler, you mustn't call anything that would cause the kernel scheduler to run. Just about the only safe call you can make from your signal handler is **release_sem()**.

The most recent alarm requested cancels any previous request. For example, in this sequence...

```
/* Ask for an alarm ten seconds from now. */
set_alarm(10e6, B_ONE_SHOT_RELATIVE_ALARM);
/* Ask for an alarm one second from now. */
set_alarm(10e5, B_ONE_SHOT_RELATIVE_ALARM);
```

...only the second alarm request will be fulfilled—the first requested is cancelled when the second **set_alarm()** call is made. This applies to all alarm types; for example, a one-shot alarm request will cancel an active periodic alarm.

To explicitly cancel the previous alarm request without installing a new alarm, do this:

```
set_alarm(B_INFINITE_TIMEOUT, B_PERIODIC_ALARM);
```

This cancels the previous alarm request regardless of the type of alarm.

---

set_signal_stack()

Declared in: be/posix/signal.be.h

void **set_signal_stack(**void ***ptr*, size_t *size***)**

Sets the location and size of the stack that's used by the thread's signal handlers.

Constants

B_INFINITE_TIMEOUT

```
B_INFINITE_TIMEOUT
```

The inifinite timeout value can be used to specify, to timeout-accepting functions, that you're willing to wait forever.

B_OS_NAME_LENGTH

```
B_OS_NAME_LENGTH
```

This constant gives the maximum length of the name of a thread, semaphore, port, area, or other operating system bauble.

B_PAGE_SIZE

```
B_PAGE_SIZE
```

The **B_PAGE_SIZE** constant gives the size, in bytes, of a page of RAM.

The Be Book The Kernel Kit The Kernel Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Miscellaneous Functions and Constants Function Summary

Function Summary

Function	Description
clear_caches()	Invalidates the instruction and data caches
debugger()	Throws the calling thread into the debugger
disable_debugger()	Causes the thread to die instead of entering the debugger on all exceptions
set_alarm()	Registers for a SIGALRM signal to be sent sometime in the future
set_signal_stack()	Sets the stack used by the signal handlers for the current thread

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimeSource

Derived from: virtual BMediaNode
Declared in: be/media/TimeSource.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BTimeSource class represents a clock to which nodes can be slaved. By slaving all your nodes to a single master time source, they can be kept in sync with each other.

If a node can, either by design or as a side benefit of the underlying hardware, provide reliable timing services, it might make sense for it to be derived from BTimeSource as well as from whatever other classes it might be derived from, such as BBufferProducer or BBufferConsumer. Doing so will allow other nodes to be slaved to your node's conception of time.

Note that although a BTimeSource is implemented as a real object (and is therefore not a purely abstract class), other nodes won't call your BTimeSource's member functions directly—instead, your BTimeSource will provide data that other nodes will then read.

There are invisible system implementations of the BTimeSource protocol that serve as stand-ins for other nodes, so if you call **`BMediaRoster::SetTimeSource()`** to make one of your nodes (which is derived from BTimeSource) a time source for some other node, the other node might see a system stand-in object, not the actual BTimeSource-derived object.

This abstraction layer serves a valuable purpose: it enforces the desire to prevent any two nodes from having to know anything about each other beyond the Media Kit protocols defined in this chapter; this sort of low-level interdependency is discouraged, because it decreases interoperability.

Keeping Time

Although it can be confusing at first, keep in mind that a node derived from both BTimeSource and BBufferProducer (or BBufferConsumer)—which is therefore a time source, as well as a producer or consumer of buffers—has to deal with two different time concepts. As a BTimeSource, it needs to understand requests in real time, while as a BBufferProducer or BBufferConsumer, it needs to accept requests in performance time.

Real time refers to the actual passage of time, as reported by **`system_time()`** or the **`BTimeSource::RealTime()`** function. It's measured in microseconds.

Performance time runs in "time units" which aren't necessarily directly related to real time. Since your code will have to deal with both kinds of time, you need to be sure to convert between the two time systems when it's necessary to do so. Use the **`BTimeSource::RealTimeFor()`** function to do this.

For example, to calculate a timeout value, given a desired performance time, and an estimated latency on the connection, you might use the following code:

```
bigtime_t timeout = TimeSource()->RealTimeFor(performance_time,
         estimated_latency) - TimeSource()->RealTime();
```

This code converts the *performance_time* into the driving time source's units, then subtracts the current real time, which results in the desired timeout value.

Hook Functions

- `HandleMessage()`
- `SnoozeUntil()`
- `SetRunMode()`
- `TimeSourceOp()`

Constructor

BTimeSource()

protected:

```
BTimeSource()
```

The standard BTimeSource constructor. You should never directly instantiate a BTimeSource; instead, you should create a node class derived from BTimeSource (and possibly other BMediaNode-derived classes as well) and use the BMediaRoster to instantiate the node as desired.

Member Functions

BroadcastTimeWarp()

```
void BroadcastTimeWarp(bigtime_t atRealTime,
    bigtime_t newPerformanceTime)
```

Whenever time jumps suddenly (for instance, when a seek operation occurs), call this function to inform all slaved nodes that time has jumped in a discontinuous manner. You should also call this function if the time from which your time source derives its time jumps.

At the real time specified by *atRealTime*, the performance time will instantaneously jump to *newPerformanceTime*.

GetStartLatency()

```
status_t GetStartLatency(bigtime_t *outLatency)
```

Returns in *outLatency* the amount of time, in microseconds, needed for the time source to start up, including the time needed to start up any slaved nodes that are started.

RETURN CODES

- B_OK. The latency value was returned successfully.
- **B_NO_INIT**. The time source has not been initialized.

GetTime()

```
status_t GetTime(bigtime_t *performanceTime, bigtime_t *realTime,
    float *drift)
```

Returns the most recently published time information for this time source; this information specifies the performance time and real time of the last published time stamp, as well as the *drift* value, which can be used to interpolate the true current performance time, given a more accurate real time, as follows:

```
bigtime_t performanceTime;
bigtime_t realTime;
float drift;

while (GetTime(&performanceTime, &realTime, &drift) != B_OK);
performanceTime = performanceTime + (RealTime() - realTime) * drift;
```

Note, however, that the resulting `performanceTime` is the same value you would have gotten by calling **`PerformanceTimeFor()`**, which you should use instead.

RETURN CODES

- B_OK. The returned information is accurate.
- Other values. The returned information is unreliable.

HandleMessage()

protected:

virtual status_t **HandleMessage**(int32 *code*, const void **message*, size_t *size*)

Given a message received on the control port, this function dispatches the message to the appropriate BTimeSource hook function. If the message doesn't correspond to a hook function, an appropriate error be returned.

If your node derives from BTimeSource, your implementation of `HandleMessage()` should call all inherited forms of `HandleMessage()`.

RETURN CODES

- B_OK. The message was dispatched.
- Other errors. The message couldn't be dispatched, possibly because it doesn't correspond to a hook function.

IsRunning()

bool **IsRunning**(void)

Returns **`true`** if the BTimeSource is currently progressing through time or **`false`** if it's stopped.

Now() , PerformanceTimeFor() , RealTimeFor()

bigtime_t **Now**(void)
bigtime_t **PerformanceTimeFor**(bigtime_t *realTime*)
bigtime_t **RealTimeFor**(bigtime_t *performanceTime*, bigtime_t *withLatency*)

`Now()` returns an approximation of what the current performance time is.

`PerformanceTimeFor()` returns an estimate of the performance time represented by the specified real time (as returned by **`BTimeSource::RealTime()`**).

`RealTimeFor()`, given a performance time, returns an approximation of the corresponding real time, adjusted by the given latency.

PublishTime()

protected:

void **PublishTime(**bigtime_t *performanceTime*, bigtime_t *realTime*, float *drift***)**

While your time source is running, you should repeatedly call this function in order to constantly refresh the mapping between real time and performance time. When your time source is stopped, you should call this function once with values of zero for all three arguments.

The arguments have the following meanings:

- *performanceTime* is the precise current performance time.
- *realTime* is the current real time.
- *drift* indicates the value which indicates the rate at which the performance time changes compared to real time.

The *drift* value makes it possible to interpolate intermediate values. For instance, if playback of a video source is progressing at normal speed, the *drift* value would be 1.0, indicating that performance time and real time progress at the same rate.

However, if the movie is playing at half-speed, *drift* would be 0.5, so that for every one unit of real time that passes, only a half-unit of performance time would pass. This information is used to compute times without having to query the time source each time an update must occur.

The data furnished by this function (which you should try to call about 20 times a second, although variations in frequency are acceptable as long as the *drift* value doesn't change much) may be the only information the rest of the world sees from your node.

RealTime()

static bigtime_t **RealTime(**void**)**

Returns the current absolute real time reference that all time sources measure themselves against. This is the only call that you should rely upon to obtain this value (don't use the Kernel Kit's **`system_time()`** function).

As of this time, `RealTime()` and **`system_time()`** do return the same value; however, you shouldn't rely upon this relationship. When doing media stuff, you should always use `RealTime()`.

RealTimeFor() see Now()

Seek()

protected:

virtual void **Seek(**bigtime_t *performanceTime*, bigtime_t *atRealTime***)**

Implement this function to handle a seek request. When a BTimeSource's performance time is adjusted, it needs to broadcast the change to all nodes slaved to it; call **BroadcastTimeWarp()** to do this.

Be sure to queue at least one seek request, so seek operations can be requested in advance. The seek request should occur at the real time specified by *atRealTime*.

SendRunMode()

protected:

void **SendRunMode(**run_mode *mode***)**

This function transmits the specified *mode* to all the nodes slaved to this node, so they know that their time source's run mode has changed. This function is called by **SetRunMode()**; it may or may not make sense to call it elsewhere, depending on your BTimeSource implementation.

SetRunMode()

protected:

virtual void **SetRunMode(**run_mode *mode***)**

This hook function is called when someone requests that your node's run mode be changed. Be sure to call through to either BMediaNode::SetRunMode() or BTimeSource::SetRunMode().

Possible values for *mode* are:

Constant	Description
BMediaNode::B_FREEWHEELING	Keep data accurate, even if the performance lags or runs too fast.
BMediaNode::B_DECREASE_PRECISION	If the performance starts to lag, try to catch up.
BMediaNode::B_INCREASE_LATENCY	If the performance starts to lag, increase playout delay so buffers are delivered less frequently.
BMediaNode::B_DROP_DATA	If the performance starts to lag, skip data.

SnoozeUntil()

virtual void **SnoozeUntil(**bigtime_t *performanceTime*, bigtime_t *withLatency* = 0,
bool *retrySignals* = false**)**

Waits until the specified *performanceTime* (specified in performance time) arrives in real time. If *withLatency* is non-zero, `SnoozeUntil()` waits until *withLatency* microseconds (in real time) prior to the specified performance time; this lets you have time for setup prior to starting some operation.

Because *performanceTime* is specified in performance time, and *withLatency* is specified in real time, you can't just subtract *withLatency* microseconds from *performanceTime* and snooze until that time, because *performanceTime* may be passing at a rate faster than or slower than real time (if, for example, the time source is running at double-speed).

If *retrySignals* is **`false`** (as is the default), `SnoozeUntil()` returns **`B_INTERRUPTED`** if it's interrupted by a signal before *performanceTime*; if *retrySignals* is true, the function will go right back to sleep and not return until the time has actually arrived.

RETURN CODES

- B_OK. No error; the snooze has completed without incident.
- **`B_MEDIA_TIME_SOURCE_STOPPED`**. The time source is stopped; snoozing on a stopped time source is a bad thing.

Start()

protected:

virtual void **Start(**bigtime_t *realTime***)**

This function is called when someone wants the BTimeSource to begin running; the *realTime* argument indicates the real time at which the time source should start. Implement the function to queue the request as necessary.

Be sure to queue at least one start request, so that start requests can be filed in advance.

A BTimeSource measures start and stop times in real time, not in performance time, because BTimeSources are the objects used to perform the mapping between real time and performance time.

Stop()

protected:

virtual void **Stop(**bigtime_t *realTime***)**

Stop() is called when someone wants the BTimeSource to stop; the realTime argument indicates the real time at which the time source should stop running. Implement the function to queue the request.

Be sure to queue at least one stop request, so stop requests can be filed in advance.

A BTimeSource measures start and stop times in real time, not in performance time, because BTimeSources are the objects used to perform the mapping between real time and performance time.

TimeSourceOp()

virtual status_t **TimeSourceOp(**const time_source_op_info &*op*,
void *_*reserved* = 0**)**

This function is called by the **BMediaRoster::Start()**, **BMediaRoster::Stop()**, and **BMediaRoster::Seek()** functions to perform the requested activities.

You must implement this function to handle these requests.

Return **B_OK** if the request was handled successfully; otherwise, return an appropriate error code.

Constants

time_source_op

Constant	Meaning
`B_TIMESOURCE_START`	Start the time source.
`B_TIMESOURCE_STOP`	Stop the time source.
`B_TIMESOURCE_STOP_IMMEDIATELY`	Stop the time source immediately.
`B_TIMESOURCE_SEEK`	Seek the time source.

These constants identify the various operations `TimeSourceOp()` will be called upon to handle.

Defined Types

time_source_op_info

```
struct time_source_op_info {
   time_source_op op;
   int32 _reserved1;
   bigtime_t real_time;
   bigtime_t performance_time;
   int32 _reserved2[6];
};
```

Describes a request to `TimeSourceOp()`.

- `op` indicates the operation to be performed.
- `real_time` indicates the time at which the request is to be filled (this is the *atRealTime* argument passed to the `BMediaRoster::SeekTimeSource()`, `StopTimeSource()`, or `StartTimeSource()` function).
- `performance_time` is the *newPerformanceTime* argument passed to `SeekTimeSource()`.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaNode

Derived from: none
Declared in: be/media/MediaNode.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BMediaNode class is the superclass of all participant nodes in the Media Kit. However, you'll never derive directly from BMediaNode; instead, you'll derive from one of the system interface classes which in turn are derived from BMediaNode. Look at the documentation for those other classes (such as BBufferProducer and BBufferConsumer) for details on how they're used, or see "Creating New Node Classes" on page16 for discussion on how this is done.

Because of the quirks of virtual inheritance (required by the use of multiple inheritance), your node's constructor will have to call the BMediaNode constructor. See "About Multiple Virtual Inheritance" on page21.

Applications shouldn't call a node's member functions directly; instead, you call the BMediaRoster with a reference to the node and let the request come to the node through the control port. The only exception is if the node is subclassed directly within the application, in which case `Acquire()`, `ID()`, `Node()`, and `Release()` can be called directly once the node is registered—if you do this, be sure you don't call them after the node is destroyed.

Calling a node's member functions directly from outside the node itself can result in the chain of functions involved in coordinating nodes to be called out of order. Worse, deadlock can result. So just don't do it, even if you think you've found a safe way to pull it off.

Creating Your Own Node

Realtime Allocators and Thread Locking

Media nodes are highly timing-sensitive creatures. The slightest delay in performing their work can cause drastic problems in media playback or recording quality. Virtual memory, normally of great benefit to users, can work against them when doing media work. A poorly-timed virtual memory hit can cause breaks in media performances.

The realtime memory allocation and locking functions provide a means for nodes to lock down their memory to prevent it from being cached to disk by the virtual memory system. This avoids situations in which the node has to pause while it or its memory is fetched back from the swap file.

The user can use the Media preference application to configure what types of nodes should use locked memory. Nodes should typically use the realtime memory allocation functions instead of `malloc()` and `free()`. `rtm_alloc()` will automatically handle locking the memory if the **`B_MEDIA_REALTIME_ALLOCATOR`** flag is set, so your node doesn't have to worry about it.

In addition, if the realtime flag corresponding to the type of node you're writing is set, your node should also call **`media_init_realtime_thread()`** to lock down the stacks of its threads. Properly-written nodes can always call **`media_init_realtime_thread()`**, without checking the realtime flags, because this function will return **`B_MEDIA_REALTIME_DISABLED`** if the corresponding flag isn't set. You can simply ignore the error and move on.

For example:

```
int32 *myThreadData = rtm_alloc(4096);
myThread = spawn_thread(myThreadFunction, "Node Thread", B_NORMAL_PRIORITY,
         &myThreadData);
status_t err = media_realtime_init_thread(myThread, 32768,
         B_MEDIA_REALTIME_VIDEO);
if (err != B_OK && err != B_MEDIA_REALTIME_DISABLED) {
   printf("Can't lock down the thread.n");
}
...
```

If your node requires realtime performance from an add-on or shared library, you can use the **`media_init_realtime_image()`** function to lock down that image in memory. Note, however, that any uses of `malloc()` by that image won't allocate locked memory; you can't control that. Still, locking down the image itself can help performance even further.

Standard BeOS system libraries are Be's responsibility. If it's appropriate for them to be locked, they're locked for you. Don't lock them yourself. Both libmedia.so and libroot.so have **`media_init_realtime_image()`** called on them.

Negotiating a Connection

Establishing a connection between two nodes is a multi-step process. The nodes need to agree upon a data format they both support before the connection can even be established.

Special Considerations

If the node you're writing could be connected by the system mixer (using the Audio preference application, for example) as the default output, the node needs to be as flexible as possible in terms of the formats it accepts on its free inputs in the `GetNextInput()` function. The format your node returns from `GetNextInput()` will be used as the starting poing in the negotiation process; the more wildcards you support, the better.

An application that wants to establish a connection between some other node and your node will determine the format from the inputs into your node and the outputs from the other node, then call `BMediaRoster::Connect()` with that format.

If there are any wildcards in the format passed to **`BMediaRoster::Format()`**, the media roster will call **`BBufferProducer::ProposeFormat()`** in the node being connected to your output node; the producer will specialize the wildcards to construct the least-specific format that will guarantee that any remaining wildcards can be specialized by your node without becoming incompatible with the producer.

The resulting format may have some wildcards left (or, if the producer is particularly picky, there may be none at all). The media roster will then pass this format to your consumer node's `BBufferConsumer::AcceptFormat()` function. This function should be implemented to specialize the remaining wildcards and return this format, which should describe a specific format. This format will be used to establish the connection.

Hook Functions

- AddOn()
- AddTimer()
- ControlPort()
- DeleteHook()
- GetNodeAttributes()
- HandleMessage()
- NodeRegistered()
- Preroll()
- RequestCompleted()
- Seek()
- SetRunMode()
- SetTimeSource()
- Start()
- Stop()
- TimeWarp()

Constructor and Destructor

BMediaNode()

protected:

```
explicit BMediaNode(const char *name)
```

Call this from your derived node's constructor.

The node is created with a reference count of 1; the count is incremented each time the `Acquire()` call is issued, and decremented each time **`Release()`** is called. When the reference count becomes zero, the node is deleted.

~BMediaNode

```
~BMediaNode()
```

You may never delete a BMediaNode because you don't know for certain when the Media Server is done with it. Instead, the Media Server maintains a reference count for the node, and when it is no longer in use, the node will be deleted automatically.

Member Functions

Acquire() , Release()

```
BMediaNode *Acquire(void)
BMediaNode *Release(void)
```

`Acquire()` returns a pointer to the node, after incrementing the node's reference count.

`Release()` releases the node by decrementing its reference count. If the count reaches zero, the node is deleted and **`NULL`** is returned; otherwise, a pointer to the node is returned.

Although you usually can't call node member functions directly from within an application, you can call `Acquire()` and `Release()` directly if the node is subclassed within the application itself (rather than in an add-on).

AddNodeKind()

```
void AddNodeKind(uint64 kind)
```

Adds a kind to the set of kinds supported by the node. This lets the system know what types of node interfaces are supported by the node's implementation. Possible values include **`B_BUFFER_PRODUCER`** (which indicates that the node implements the BBufferProducer protocol) and **`B_PHYSICAL_INPUT`** (which indicates that the node implements a physical input, such as a sound digitizing input device). For a complete list of kind values, see "node_kind" on page182.

In general, you don't need to call this function. The base system classes call `AddNodeKind()` automatically to set up the node type flags; for example, a BBufferProducer automatically calls `AddNodeKind(B_BUFFER_PRODUCER)`. The only time it's necessary go call AddNodeKind() is if the node you're implementing is a physical device or a mixer, in which case you need to add the **`B_PHYSICAL_INPUT`**, **`B_PHYSICAL_OUTPUT`**, or **`B_SYSTEM_MIXER`** flag.

AddOn()

```
virtual BMediaAddOn *AddOn(int32 *outInternalID) const = 0
```

Implement this function to return a pointer to the BMediaAddOn that instantiated the node. If the node lives in an application (rather than in an add-on), return **`NULL`**. If the node is in an add-on, *outInternalID* should be changed to contain the internal ID number of the node within the add-on.

AddTimer() , TimerExpired()

```
virtual status_t AddTimer(bigtime_t toPerformanceTime, int32 cookie)
void TimerExpired(bigtime_t notifyPoint, int32 cookie, status_t error = B_OK)
```

Your node should implement the `AddTimer()` function to remember the *cookie* and time given. When the time *toPerformanceTime* is reached, your node should call `TimerExpired()` with the corresponding *cookie* value, passing the recorded *toPerformanceTime* value as the *notifyPoint* argument. This will, in turn, cause the `BMediaRoster::SyncToNode()` call that instigated the timer to return to the caller.

Your implementation of `AddTimer()` should return `B_OK` if all is well; otherwise it should return an appropriate error code.

ControlPort()

```
virtual port_id ControlPort(void) const = 0
```

Returns the port_id of the port to which the node listens for requests. Your node must implement this to return a valid Kernel Kit port.

DeleteHook()

protected:

```
virtual status_t DeleteHook(BMediaNode *node)
```

The `DeleteHook()` function is called to delete the BMediaNode object. You may augment this if you need to perform additional work before the node is deleted, but you should always either include the line:

```
delete this;
```

or you should call through to the inherited form of the function. Return `B_OK` if the node was deleted successfully, otherwise return an appropriate error code.

GetNodeAttributes()

```
virtual status_t GetNodeAttributes(media_node_attribute *outAttributes,
    size_t inMaxCount)
```

Implement this function to fill the *outAttributes* array (which has room for *inMaxCount* attributes) with your node's attributes.

Return **B_OK** if all is well, or return an appropriate error code.

HandleBadMessage()

void **HandleBadMessage(**int32 *message*, const void **data*, size_t *size***)**

If your node receives a message that neither the node, nor any interface from which the node is derived, understands the message, pass the message along to this function, which will work magic to deal with the problem one way or another. All arguments received by the **HandleMessage()** function should be passed directly through to `HandleBadMessage()`.

HandleMessage()

virtual status_t **HandleMessage(**int32 *message*, const void **data*, size_t *size***)**

Given a message received on the control port, this function dispatches the message to the appropriate BMediaNode hook function. If the message doesn't correspond to a hook function, **B_ERROR** is returned.

When you implement a media node of your own (derived from BBufferConsumer, BBufferProducer, etc), you always need to call through to `BMediaNode::HandleMessage()` from your node's implementation of `HandleMessage()`. This is crucial, to be sure that every ancestor of your node gets to look at the message and attempt to process it.

For example, if your node inherits from both BBufferProducer and BBufferConsumer, you should call **BBufferProducer::HandleMessage()** and **BBufferConsumer::HandleMessage()**, then `BMediaNode::HandleMessage()`, like this:

```
virtual status_t MyBufferProducerConsumer::HandleMessage(int32 message,
         const void *data, size_t size) {
   if (message == SOME_THING_I_DO) {
      DoWhatever();
   }
   else if (BBufferConsumer::HandleMessage(message, data, size) &&
            BBufferProducer::HandleMessage(message, data, size) &&
            BMediaNode::HandleMessage(message, data, size)) {
      BMediaNode::HandleBadMessage(message, data, size);
   }
}
```

Note that **BMediaNode::HandleBadMessage()** is called if none of the `HandleMessage()` implementations accept the message.

Values of *message* between 0x60000000 and 0x7FFFFFFF are available for use by applications. Values below 0x60000000 are reserved for use by the Media Kit, and typically correspond to specific virtual hook functions within your node. If you can show just cause for needing to

know the *message* value for a particular hook, you can try emailing devsupport@be.com and see if we agree with you, in which case we may share that information.

Don't reverse-engineer the *message* values; if you really need to know, ask us. Otherwise, we won't know that a particular message code number shouldn't be changed. In general, it's a bad idea to rely on specific values, although there may be cases in which it's necessary.

RETURN CODES

- B_OK. The message was dispatched.
- **B_ERROR**. The message couldn't be dispatched, possibly because it doesn't correspond to a hook function.

HitSyncPoint() see Sync()

ID()

```
media_node_id ID(void) const
```

Returns the media_node_id assigned to the node by the Media Server. The result is 0 if the node hasn't been registered yet, and negative if an error occurred while attempting to register the node.

Kinds()

```
uint64 Kinds(void) const
```

Returns a bit mask indicating what interfaces the node implements. See "node_kind" on page182 for a list of valid interface kinds.

Name()

```
const char *Name(void) const
```

Returns a human-readable string specifying the node's name. This pointer is only valid until you **Release()** the node; after that, the pointer may point into empty space.

NewChangeTag()

protected:

```
static int32 NewChangeTag(void)
```

This function, intended primarily for use by BBufferConsumer nodes, creates and returns a new change tag value.

Node()

media_node **Node(**void**)** const

Returns the **media_node** structure that will be used by an application when accessing this node via the media roster.

NodeRegistered()

virtual void **NodeRegistered(**void**)**

The Media Server calls this hook function after the node has been registered.

NodeStopped()

status_t **NodeStopped(**bigtime_t *whenPerformanceTime***)** const

When you've finished handling a stop request (buffers will no longer be flowing), call this function. If anyone is listening for stop notifications from you, they'll be notified. The *whenPerformanceTime* argument should be the performance time of the stop command that was handled.

Anyone listening for node stop messages will be notified; this lets applications running in offline (rendering) mode know when the node has actually completed its work.

If your node is a BBufferProducer, downstream consumers will be notified that your node stopped (automatically, no less) through the **BBufferConsumer::ProducerDataStatus(B_PRODUCER_STOPPED)** call. This lets offline rendering nodes know when each of their inputs have no more data to send for the current roll.

This is especially important for nodes that can be run in **B_OFFLINE** mode.

RETURN CODES

- B_OK. No error.
- Other errors. Unable to communicate with the Media Server, or an error occurred communicating with other nodes.

Preroll()

protected:

virtual void **Preroll**(void)

This hook function may be called before your node receives a **Start()** message if the application using the node calls **BMediaRoster::PrerollNode()**. This gives the node a chance to prepare the media so that when the media is started, the response is as fast as possible.

Release() see Acquire()

ReportError()

protected:

status_t **ReportError**(node_error *whichError*,
const BMessage **info* = NULL)

Transmits the error code specified by *whichError* to anyone that's receiving notifications from this node (see **BMediaRoster::StartWatching()** and **BMediaRoster::StopWatching()** on). If *info* isn't **NULL**, it's used as a model message for the error notification message.

RETURN CODES

- B_OK. The error report was sent without error.
- BMessage errors. The message couldn't be sent.

RequestCompleted()

protected:

virtual status_t **RequestCompleted**(const media_request_info &*info*)

This function is called whenever a request issued by the node is completed. The *info* structure describes the results of the request.

The `change_tag` field in the *info* structure identifies the request that has been completed; this is the same value passed into the function that initiated the request.

Return **B_OK** if you're happy, otherwise return an appropriate error code.

RunMode() , SetRunMode()

run_mode **RunMode**(void) const

protected:

virtual void **SetRunMode(**run_mode *mode***)**

`RunMode()` returns the node's current run_mode setting.

The `SetRunMode()` hook function is called when someone requests that your node's run mode be changed.

See also: "run_mode" on page182

Seek()

protected:

virtual void **Seek(**bigtime_t *mediaTime*, bigtime_t *performanceTime***)**

This hook function is called when a node is asked to seek to the specified *mediaTime* by a call to the BMediaRoster.

The specified *performanceTime*, the time at which the node should begin the seek operation, may be in the future.

Your node is required to queue at least one each of start, stop, and seek requests, so that applications can establish, for example, both the start and stop time without having to monitor your node's progress. The actual size of these three queues is up to you. When the specified time arrives, the request should be filled.

A *mediaTime* value of 0 indicates the beginning of the media data.

Set see RunMode()

SetTimeSource() , TimeSource()

virtual void **SetTimeSource(**BTimeSource ***timeSource***)**

BTimeSource ***TimeSource(**void**)** const

The `SetTimeSource()` hook function is called when someone has requested that the node be slaved to a new time source. Augment this function to make whatever adjustments you need to make to operate at the new time scale.

`TimeSource()` returns a pointer to the BTimeSource to which the node is currently slaved. If no time source has been explicitly requested, the system time source is in use, and that's what gets returned.

The BTimeSource object returned by `TimeSource()` is only valid until the next call to **`HandleMessage()`** on that object. Therefore, if your node runs more than one thread, you need to

serialize calls to `TimeSource()` (as well as usage of the returned objects) with calls to **`HandleMessage()`**. This isn't a problem if you follow the recommended policy of running a single thread that monitors the service port with **`read_port_etc()`** and calls **`HandleMessage()`** only when a message is actually received.

Start()

protected:

virtual void **`Start(`**bigtime_t *performanceTime*`)`

This hook function is called when a node is started by a call to the BMediaRoster.

The specified *performanceTime*, the time at which the node should start running, may be in the future.

Your node is required to queue at least one each of start, stop, and seek requests, so that applications can establish, for example, both the start and stop time without having to monitor your node's progress. The actual size of these three queues is up to you. When the specified time arrives, the request should be filled.

Stop()

protected:

virtual void **`Stop(`**bigtime_t *performanceTime*, bool *immediate*`)`

This hook function is called when a node is stopped by a call to the BMediaRoster.

The specified *performanceTime*, the time at which the node should stop, may be in the future.

If immediate is true, your node should ignore the performanceTime value and synchronously stop performance. When `Stop()` returns, you're promising not to write into any BBuffers you may have received from your downstream consumers, and you promise not to send any more buffers until **`Start()`** is called again.

Your node is required to queue at least one each of start, stop, and seek requests, so that applications can establish, for example, both the start and stop time without having to monitor your node's progress. The actual size of these three queues is up to you. When the specified time arrives, the request should be filled.

Nodes must recycle all buffers they may be holding onto when they're stopped.

TimerExpired() see AddTimer()

TimeSource() see SetTimeSource()

TimeWarp()

protected:

```
virtual void TimeWarp(bigtime_t atRealTime, bigtime_t newPerformanceTime)
```

This hook function is called when the time source to which the node is slaved is repositioned (via a seek operation) such that there will be a sudden jump in the performance time progression as seen by the node. The *newPerformanceTime* argument indicates the new performance time; the change should occur at the real time specified by the *atRealTime* argument.

The node should respond to this call by preparing for this change, so a serious stutter, failure, or acceleration in performance doesn't occur. Appropriate measures should be taken to minimize the impact on the performance quality; for example, a segment of the sound could be looped or skipped smoothly.

Your implementation of `TimeWarp()` should call through to `BMediaNode::TimeWarp()` as well as all other inherited forms of `TimeWarp()`.

WaitForMessage()

protected:

```
status_t WaitForMessage(bigtime_t waitUntil, uint32 flags = 0,
    void *_reserved_ = NULL)
```

This function waits until either real time specified by *waitUntil* or a message is received on the control port.. The *flags* are currently unused and should be 0.

When a message is received, the appropriate HandleMessage() calls are made given the class derivation of the node:

- **`BMediaNode::HandleMessage()`** is always called first.
- If the node is derived from BBufferProducer, and the message hasn't been handled yet, **`BBufferProducer::HandleMessage()`** is called.
- If the node is derived from BBufferConsumer, and the message hasn't been handled yet, **`BBufferConsumer::HandleMessage()`** is called.
- If the node is derived from BFileInterface, and the message hasn't been handled yet, **`BFileInterface::HandleMessage()`** is called.
- If the node is derived from BControllable, and the message hasn't been handled yet, **`BControllable::HandleMessage()`** is called.

- If the node is derived from BTimeSource, and the message hasn't been handled yet, **`BTimeSource::HandleMessage()`** is called.
- If the message still hasn't been handled, the most-derived interface's **`HandleMessage()`** function is called.
- If the message hasn't been handled, **`HandleBadMessage()`** is called.

Once this has been done, `WaitForMessage()` returns. As you can see, this can be called from your control port to handle much of the work of processing received messages.

RETURN CODES

- B_OK. A message has occurred within the given time period.
- **`B_TIMED_OUT`**. The time waitUntil has arrived without a message being received.

Constants

node_error

Declared in: be/media/MediaNode.h

Constant	Description
`B_NODE_FAILED_START`	The node failed on a `Start()` request.
`B_NODE_FAILED_STOP`	The node failed on a `Stop()` request.
`B_NODE_FAILED_SEEK`	The node failed on a `Seek()` request.
`B_NODE_FAILED_SET_RUN_MODE`	The node's run_mode couldn't be set.
`B_NODE_FAILED_TIME_WARP`	The node couldn't fulfill a time warp request.
`B_NODE_FAILED_PREROLL`	The node failed on a `Preroll()` request.
`B_NODE_FAILED_SET_TIME_SOURCE_FOR`	The node's time source couldn't be changed.
`B_NODE_IN_DISTRESS`	The node is suffering in general.

The node_error type defines the errors a node can transmit to BMessengers that have registered to watch the node.

node_kind

Declared in: be/media/MediaDefs.h

Constant	Description
`B_BUFFER_PRODUCER`	The flavor implements BBufferProducer.
`B_BUFFER_CONSUMER`	The flavor implements BBufferConsumer.
`B_TIME_SOURCE`	The flavor implements BTimeSource.
`B_CONTROLLABLE`	The flavor implements BControllable.
`B_FILE_INTERFACE`	The flavor implements BFileInterface.
`B_ENTITY_INTERFACE`	The flavor implements BEntityInterface (not available for R4).
`B_PHYSICAL_INPUT`	The flavor represents a physical input point (such as an input jack).
`B_PHYSICAL_OUTPUT`	The flavor represents a physical output point (such as line output).
`B_SYSTEM_MIXER`	The flavor represents the system mixer.

Defines the type or types of node that a particular flavor supports. Note that a flavor can implement more than one type of node, so you can combine these values using a bitwise or.

run_mode

Declared in: be/media/MediaNode.h

Constant	Description
B_OFFLINE	Keep data accurate, even if the performance lags or runs too fast. This is typically used when rendering to disk.
B_RECORDING	Time-stamped buffers are being received from a node capturing them from the real world; these buffers are guaranteed to have a time stamp in the past (they're always "late").
B_DECREASE_PRECISION	If the performance starts to lag, try to catch up.
B_INCREASE_LATENCY	If the performance starts to lag, increase playout delay so buffers are delivered with less time to spare before they're needed.
B_DROP_DATA	If the performance starts to lag, skip data.

The run_mode type indicates how a node should cope if its performance rate deviates from the desired rate.

B_OFFLINE

When in offline mode (B_OFFLINE), the node doesn't need to worry about processing buffers at any particular time. Each buffer's performance time should be derived from the time stamped on the buffer, rather than from a BTimeSource. In fact, you'll usually want to call set_thread_priority() to set your node's processing threads to a low priority while the node is in offline mode. This lets software render media to disk in an efficient manner, letting the user continue to work while the render occurs in the background.

B_RECORDING

Recording mode (B_RECORDING) should be used when data is being sampled from a physical input device. These devices always deliver buffers whose time stamps are in the past (they're stamped with the time at which they were sampled, which is of course in the past, unless you've stolen a time machine from a professor from the 27th century, in which case you're probably running BeOS R127.1 and this book is woefully obsolete).

Using B_RECORDING mode serves to warn other nodes that the time stamps will be in the past.

B_DECREASE_PRECISION

In B_DECREASE_PRECISION mode, your node should attempt to catch up if it falls behind, by playing buffers of media data faster than normal. For audio, this might mean playing back at a higher sampling rate; for video, the frame rate might be temporarily boosted.

B_INCREASE_LATENCY

If your node gets behind in the B_INCREASE_LATENCY run mode, your node should increase its internal latency measurement and send call the LateNoticeReceived() function in anyone above your node in the media stream.

Your node should then try to produce each buffer earlier before the buffer's performance time from that point on, so there's more time for the buffers to reach their destination.

This mode is intended to compensate for data streams in which throughput can vary over time. For example, if media data is being streamed over a network, traffic fluctuations may require your node to adapt by adding more buffering (latency).

B_DROP_DATA

When in `B_DROP_DATA` mode, your node should simply skip buffers if if falls behind. Note that you still receive the buffers, but you should ignore any that you must in order to keep playing as many buffers as possible at the correct performance times.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaNode Function Summary

Function Summary

Function	Description
Acquire()	Returns a reference to the node
AddNodeKind()	Adds a kind to the set of kinds supported by the node
AddOn()	Returns the BMediaAddOn that instantiated the node
AddTimer()	Hook: implement to remember a time that should be reported
ControlPort()	Hook: return the port_id of the node's control port
DeleteHook()	Hook: called when the node is to be deleted
GetNodeAttributes()	Hook: implement to return the node's attributes
HandleBadMessage()	Copes with messages the node doesn't understand
HandleMessage()	Hook: dispatch a Media Kit message
ID()	Returns the node's ID
Kinds()	Returns a bit mask indicating what interfaces the node implements
BMediaNode()	Constructor
Name()	Returns the node's name
NewChangeTag()	Returns a new change tag
Node()	Returns a reference to the node for use when calling BMediaRoster

NodeRegistered()	Hook: called when the node is registered with the Media Kit
NodeStopped()	Informs interested parties when you've finished handling a stop request
Preroll()	Hook: gives the node a chance to prepare the media for use
Release()	Releases the node
ReportError()	Transmits an error notification to anyone that's watching
RequestCompleted()	Hook: called when requests have been completed or have failed
RunMode()	Returns the node's run_mode
Seek()	Hook: seeks to the specified time in the media
SetRunMode()	Hook: sets the node's run mode
SetTimeSource()	Hook: specifies the time source the node should be synchronized to
Start()	Hook: specifies a time at which the node should start
Stop()	Hook: stops the node
TimeSource()	Returns the BTimeSource to which the node is slaved
TimeWarp()	Hook: informs the node of a sudden change in its time source's time
TimerExpired()	Notify interested parties when a notify point is reached
WaitForMessage()	Waits for the given amount of time,

	or until a message is received.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferProducer

Derived from: virtual BMediaNode
Declared in: be/media/BufferProducer.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A BBufferProducer is a BMediaNode that emits buffers containing media data that other nodes (BBufferConsumers in particular) will receive and, potentially, process. If your node wants to emit buffers, it must be derived from BBufferProducer and override the hook functions to implement the BBufferProducer protocol.

Video Clipping

Currently, the only video clipping format supported by the Media Kit is **`B_CLIP_SHORT_RUNS`**, although there is a function in this class for converting between this format and BRegions.

This format begins with a header, consisting of two int16 values:

offsetX	X offset for all following coordinates.
offsetY	Y offset for all following coordinates.

These values indicate the offset for the X and Y coordinates indicated throughout the rest of the clipping data.

The remainder of the clipping data consists of entries indicating each line of video data, as follows:

numShorts	The number of values in the *coordList*. Always an even number. If negative, repeats the previous entry |*numShorts*| times.
coordList...	List of coordinates. Even entries are left-edge X coordinates, odd entries are right-edge X coordinates.

The clipping data contains one of these entries for each time the clipping information changes.

For example, if the clipping is a rectangle with the left edge at 100, top edge at 50, right edge at 300, and bottom edge at 200, the clipping data for a 640x480 display might be:

header
  offsetX: 0
  offsetY: 50

entry 1
  numShorts: 2
  coordList: 100, 300

entry 2
  numShorts: -150

entry 3
  numShorts: 2
  coordList: 0, 639

entry 4
  numShorts: -280

The header indicates that the clipping data begins at row 50.

The first entry indicates that clipping should span from column 100 to column 300 on the first row of clipping (row 50). The second entry says to repeat this 150 times.

Entry 3 indicates that clipping from that point on should be from column 0 to column 639 (the entire width of the display). Entry 4 causes this to repeat 280 times, to the bottom of the display.

Seek Tags

In order to support media formats that don't provide timing information in their outer encapsulation layer, or to provide enhanced seeking performance for media formats that support key frames, the Media Kit supports the concept of seek tags. Producers that know their data doesn't have timing information, or that can provide enhanced seeking using special tags, should put a tag in the `user_data` field of the buffer headers it sends. This tag can contain any data the producer wants.

Consumers that can derive good timing information from these packets after decoding them should then choose appropriate seek points (usually key frames) and cache the performance time and tag values of the first buffer that arrives at that seek point.

Producers that can't seek without help from the decoder can then query the consumer by calling **`FindSeekTag()`**. This causes the consumer's **`SeekTagRequested()`** function to be called. This returns the seek tag and time that are closest to the requested time. The producer can then use this information locate the appropriate point in the media data.

The easiest way to use this is to use the file offset as the tag data, but any value that makes sense to the producer can be used, since the consumer just saves a copy of the data and passes it back without looking at it.

Time	Seek Tag
0.0 seconds	0
0.1 seconds	<none>
0.2 seconds	2
0.3 seconds	<none>

In this simple example, we have four buffers, two of which have seek tags recorded (at 0.0 seconds and 0.3 seconds). If the producer is seeking to 0.2 seconds, it would call **`FindSeekTag()`**, like this:

```
media_seek_tag tag;
bigtime_t time;
FindSeekTag(&destination, 0.2*1000000, &tag, &time);

/* now we can use the tag to seek */
```

If the tag contains a file offset, we can simply seek to that offset in the file and we're ready to go.

In this example, the returned tag is "2" and the time is 0.2 seconds, because there's a seek tag located precisely at the requested time. However, if we look for a seek tag for 0.1 seconds, we get a returned tag of "0" and a time of 0.0 seconds, because that's the closest matching tag to the requested time.

Hook Functions

- AdditionalBufferRequested()
- Connect()
- Disconnect()
- DisposeOutputCookie()
- EnableOutput()
- FormatChangeRequested()
- FormatProposal()
- FormatSuggestionRequested()
- GetLatency()
- GetNextOutput()
- HandleMessage()
- LatencyChanged()
- LateNoticeReceived()
- PrepareToConnect()
- SetBufferGroup()
- SetPlayRate()
- VideoClippingChanged()

Constructor

BBufferProducer()

protected:

explicit **BBufferProducer(**media_type *producerType***)**

Constructs the BBufferProducer object. The *producerType* specifies the type of media data that will be output by the node.

If the node will produce more than one type of data, your BBufferProducer subclass should set the kind to the default (which is a wildcard value).

If your node has additional latency on startup, you should call **SetInitialLatency()** to record this information. This might be the case if the buffers your node produces are created from an input signal which refreshes infrequently, such as a television signal.

In BeOS Release 4.5.2 and earlier, the *producerType* has a default value; it no longer does, and you'll have to specify the media type yourself.

Member Functions

AdditionalBufferRequested()

```
virtual status_t AdditionalBufferRequested(const media_source &source,
    media_buffer_id previousBufferID,
    bigtime_t previousTime,
    const media_seek_tag *previousTag)
```

When a consumer calls `BBufferConsumer::RequestAdditionalBuffer()`, this function is called as a result. Its job is to call `SendBuffer()` to immediately send the next buffer to the consumer.

The *previousBufferID*, *previousTime*, and *previousTag* arguments identify the last buffer the consumer received. Your node should respond by sending the next buffer after the one described.

The *previousTag* may be **NULL**.

Return `B_OK` if all is well; otherwise return an appropriate error code.

ChangeFormat()

```
status_t ChangeFormat(const media_source &source,
    const media_destination &destination,
    media_format *newFormat)
```

Informs the *destination* that the data flowing between *source* and *destination* is immediately changing to the format specified by *newFormat*.

You must never call `SendBuffer()` while this call is pending.

RETURN CODES

- B_OK. The format change request has been sent without error.
- **B_MEDIA_CHANGE_IN_PROGRESS**. A mutual exclusion error has occurred with `SendBuffer()`; `ChangeFormat()` and `SendBuffer()` can't both be running at the same time.
- Other errors. You may receive other errors if the consumer doesn't agree with the new format you're requesting.

ClipDataToRegion()

```

```

```
static status_t ClipDataToRegion(int32 format, int32 byteCount,
    const void *clipData, BRegion *region)
```

Given *byteCount* bytes of clipping data *clipDatain* the specified *format*, makes the specified *region* match the clipping region.

The *region* you specify must already exist.

The only format currently supported is **B_CLIP_SHORT_RUNS**.

RETURN CODES

- B_OK. The clip data was converted without error.
- **B_MEDIA_BAD_CLIP_FORMAT**. The specified clip format is invalid.

See also: **BBufferConsumer::RegionToClipData()**

Connect()

```
virtual void Connect(status_t status, const media_source &source,
    const media_destination &destination,
    const media_format &format,
    char *ioName) = 0
```

Implement this hook function to establish a connection between the source and the destination. The format negotiation is already complete by the time `Connect()` is called, so you have to accept the specified *format*.

The *status* argument indicates whether or not the connection actually took place; this is the result code returned by the **BBufferConsumer::Connected()** function or an error code indicating an error that has occurred during other preparation for the connection.

If status isn't **B_OK**, you should release the media_source that was reserved for this connection by **PrepareToConnect()**; this lets it be used by other connection attempts.

On entry, *ioName* contains the connection name specified by the consumer (this may be different from the name specified by the **BBufferProducer::PrepareToConnect()** function). On return, *ioName* should point to a name for the connection; if the name really matters to you, copy the name you want the connection to have back into *ioName*; otherwise, you can leave it alone.

Disconnect()

```
virtual void Disconnect(const media_source &source,
    const media_destination &destination) = 0
```

Your implementation of `Disconnect()` should terminate the connection between the specified *source* and *destination*. Once you return from this function, you shouldn't send any further buffers on the connection.

If a BBufferGroup has been specified for your producer (via the **`SetBufferGroup()`** function), you should delete it here.

DisposeOutputCookie()

```
virtual status_t DisposeOutputCookie(int32 cookie) = 0
```

Once a client has finished iterating through your outputs via **`GetNextOutput()`** calls, it will call this function with the last value you returned as a *cookie*. This gives you the opportunity to dispose of any memory you may have allocated for the iteration process.

Return **`B_OK`** if the cookie is successfully disposed of (or if nothing needs to be done); otherwise, return an appropriate error code.

EnableOutput()

```
virtual void EnableOutput(const media_source &whichOutput,
    bool enabled,
    int32 *_depreciated_)
```

This hook function is called when a consumer's **`SetOutputEnabled()`** function is called. This indicates whether or not the output specified by *whichOutput* needs to be sent buffers. You must implement this function so that you don't send buffers to outputs that don't need them. The *_depreciated_* argument is no longer used.

By default, output is enabled.

FindLatencyFor()

```
status_t FindLatencyFor(const media_destination &forDestination,
    bigtime_t *outLatency,
    media_node_id *outTimeSource)
```

`FindLatencyFor()` returns the latency introduced by sending data to the destination *forDestination*. On return, the latency will be stored in *outLatency*, and the time source used by *forDestination* will be available in *outTimeSource* (unless an error is returned, in which case these values are undetermined).

The latency of sending a buffer from one time source to another should always be assumed to be zero, since there may be no relationship between the progress of time of two different time sources.

RETURN CODES

- B_OK. The latency was returned without error.
- **B_MEDIA_BAD_DESTINATION**. The destination is invalid.
- Port errors. The request couldn't be sent to the destination.

FindSeekTag()

```
status_t FindSeekTag(const media_destination &forDestination,
    bigtime_t inTargetTime,
    media_seek_tag *outTag, bigtime_t *outTaggedTime,
    uint32 *outFlags = 0,
    uint32 inFlags = 0)
```

In order to improve seek performance, the Media Kit provides the concept of **seek tags**. These are special tags that identify easily-located points in media data (such as key frames in Cinepak video). The `FindSeekTag()` function asks the consumer specified by *forDestination* for the nearest seek tag to the time specified by *inTargetTime*, and returns the tag in *outTag* and the time corresponding to that tag in *outTaggedTime*. On return, *outFlags* (if the pointer isn't **NULL**) contains flags giving further details about the tag.

There are currently no defined values for *inFlags* or *outFlags*.

RETURN CODES

- B_OK. No error.
- Port errors. An error occurred communicating with the Media Server.

FormatChangeRequested()

```
virtual status_t FormatChangeRequested(const media_source &source,
    const media_destination &destination,
    media_format *ioFormat,
    int32 *_depreciated_) = 0
```

Implement `FormatChangeRequested()` to change the format of the media data flowing from the given *source* to the specified *destination* to the format specified by *ioFormat*. If there are wildcards specified in *ioFormat*, fill them in to match the format you prefer before returning from this call. You should ignore the *_depreciated_* argument; it's no longer used.

This call is issued synchronously by the destination, so you can't ask it if the format is acceptable. Fortunately, since the destination issued the request, you can safely assume that it's fine.

Return **B_OK** if the change request is processed successfully; otherwise, return an appropriate error code.

See also: **FormatSuggestionRequested()**, **FormatProposal()**

FormatProposal()

```
virtual status_t FormatProposal(const media_source &output,
      media_format *format) = 0
```

Your BBufferProducer should implement this function to verify that the proposed media_format is suitable for the specified *output*. If any fields in the *format* are wildcards, and you have a specific requirement, adjust those fields to match your requirements before returning.

Return **B_OK** if the proposed *format* is acceptable; once you've done so, the Media Kit will assume that any connection request made on *output* with the specified format (after any changes you may have made) will succeed.

If *output* isn't available, return **B_BAD_SOURCE**.

If *format* isn't reasonable, return **B_BAD_MEDIA_FORMAT**.

FormatSuggestionRequested()

```
virtual status_t FormatSuggestionRequested(media_type type,
      int32 quality, media_format *format) = 0
```

You must implement `FormatSuggestionRequested()` to return fill the buffer pointed to by *format* that your producer is capable of emitting that meets the desired *type* and *quality* requirements.

If your producer can work with a range of possible formats, let the *quality* argument guide your selection. For example, you might choose to use 10 fps for previews, and 60 fps interlaced 640x480 for full-quality video.

If *type* is a media class that your producer doesn't want to work with, return **B_BAD_MEDIA_FORMAT**. If you're preapared to accept a wide range of values for some specific field, set that field to the wildcard value (see **media_audio_format::wildcard()** and **media_video_format::wildcard()** for more information.

Return **B_OK** if the *format* is successfully returned.

GetLatency()

```
virtual status_t GetLatency(bigtime_t *outLatency)
```

Implement this hook function to store, in *outLatency*, the total amount of latency your BBufferProducer incurs from receiving a buffer of data until it reaches its ultimate destination.

Call `FindLatencyFor()` on whatever outputs the data is being forwarded to, add your own latency to the largest of those values, and return that value.

The default implementation of `GetLatency()` finds the maximum latency of your currently-available outputs by iterating over them, and returns that value in *outLatency*; therefore, your implementation of this function may simply need to call the inherited version of this function, then add your own processing latency to the returned value.

RETURN CODES

- B_OK. The latency has been returned successfully.
- Other errors. Unable to calculate the latency.

GetNextOutput()

```
virtual status_t GetNextOutput(int32 *cookie, media_output *outOutput) = 0
```

Implement this function to return information about your available outputs. The first time it's called for a new iteration loop, the value pointed to by *cookie* will be 0. Each time `GetNextOutput()` is called, you should set it to some value that makes sense to you so you can keep track of where in the iteration process the client is, but never set it to 0.

For each call to `GetNextOutput()`, including the first, you should return one of your outputs that the client hasn't seen during the iteration loop in *outOutput*.

Once all outputs have been reported, you should return `B_ERROR`.

HandleMessage()

```
virtual status_t HandleMessage(int32 message, const void *data, size_t size)
```

When your node derived from BBufferProducer receives a message on its control port, you should handle it yourself if you know how, or dispatch to each ancestor class in turn (starting with `BBufferProducer::HandleMessage()`) until one of the `HandleMessage()` implementations returns `B_OK`. If none of the inherited implementations of this function returns `B_OK`, you should pass the message to `BMediaNode::HandleBadMessage()` to be dealt with.

Your port-listening thread should call `HandleMessage()` to dispatch the received data.

See also: `BMediaNode::HandleMessage()`, "About Multiple Virtual Inheritance" on page21

LatencyChanged()

virtual void **LatencyChanged(**const media_source &*source*, const media_destination &*destination*, bigtime_t *newLatency*, uint32 *flags***)**

This hook function is called when a BBufferConsumer that's receiving data from you determines that its latency has changed. It will call its **`BBufferConsumer::SendLatencyChange()`** function, and in response, the Media Server will call your `LatencyChanged()` function.

The *source* argument indicates your output that's involved in the connection, and *destination* specifies the input on the consumer to which the connection is linked. *newLatency* is the consumer's new latency. The *flags* are currently unused.

Override this function to implement whatever functionality you need to adjust your own latency calculations to keep the data flowing smoothly.

LateNoticeReceived()

`virtual void LateNoticeReceived(const media_source &`*`whichSource`*`,`
`    bigtime_t `*`howLate`*`, bigtime_t `*`performanceTime`*`) = 0`

This hook function is called when a BBufferConsumer that's receiving data from you determines that data is arriving late (when the **`BBufferConsumer::NotifyLateProducer()`** function is called); the exact degree to which your buffers are late is specified by the *howLate* argument. Your implementation of this function should take whatever steps are necessary to correct the problem, either by asking nodes upstream from you to deliver buffers earlier, dropping buffers, or other appropriate actions, depending on the current run mode.

The *performanceTime* argument specifies the performance time at which the notification was sent.

See also: **`BMediaNode::RunMode()`**

PrepareToConnect()

virtual status_t **PrepareToConnect(**const media_source &*whichSource*,
    const media_destination &*whichDestination*,
    media_format **format*,
    media_source **outSource*,
    char **outName***)** = 0

The `PrepareToConnect()` hook is called before a new connection between the source *whichSource* and the destination *whichDestination* is established, in order to give your producer one last chance to specialize any wildcards that remain in the *format* (although by this point there shouldn't be any, you should check anyway).

Your implementation should, additionally, return in *outSource* the source to be used for the connection, and should fill the *outName* buffer with the name the connection will be given; the consumer will see this in the `outInput->name` argument specified to

BBufferConsumer::Connected(). If your node doesn't care what the name is, you can leave the *outName* untouched.

Your `Connect()` function may return a different media_source value in *outOutput*'s *source* field than the one specified as the source argument to this function. One reason you might do this is if you implement one media_source to accept connection requests, then create a new media_source to actually handle each connection.

Return B_OK if the connection process should proceed, or an appropriate error code if something's wrong.

If you return B_OK, the consumer's Connected() function will be called, to let it know that a new connection is being established. Finally, the producer's Connect() function is called to complete the exchange.

See also: BBufferConsumer::Connected(), Connect()

ProducerType()

```
media_type ProducerType(void)
```

Returns the media_type of the media data produced by the node.

ProposeFormatChange()

```
status_t ProposeFormatChange(media_format *format,
    const media_destination &forDestination)
```

Call this function to determine whether or not the destination *forDestination* is prepared to accept buffers in the specified *format*. This function can be especially useful if you want to test various formats to select the best compatible format during a hookup request in which the requested format contains wildcards.

RETURN CODES

- B_OK. The proposed format is acceptable to the destination.
- Other errors. The proposed format is unacceptable, or an error occurred in querying the destination node.

SendBuffer()

```
status_t SendBuffer(BBuffer *buffer, media_destination &destination)
```

Call this function to send a *buffer* of media data to the specified *destination*, which must already be connected to one of your outputs. This is how your BBufferProducer object will send data downstream to BBufferConsumers to which it's connected.

It's your responsibility to ensure that the buffer's header and the data contained in the buffer itself are valid, although `SendBuffer()` will automatically fill out the following header fields for you:

- buffer
- for_id
- change_tag

In particular, be sure that if you're outputting video buffers you set the **`media_video_buffer`** to describe the video properly. If you don't, things will go badly for you.

You can obtain a buffer to fill and send by calling **`BBufferConsumer::RequestBuffer()`** on a BBufferConsumer that you own (and that's okay to use for buffers going to the specified *destination*).

RETURN CODES

- B_OK. The buffer was sent without error.
- Port errors. An error occurred sending the buffer.

SendDataStatus()

```
status_t SendDataStatus(int32 status,
    media_destination &destination,
    bigtime_t atTime)
```

Call this function to inform the specified destination whether or not there's data available from your producer node. Specify the appropriate status flag as the *status* argument, and the time at which the status takes effect as the *atTime* argument.

Possible values for the *status* argument are:

Constant	Description
`B_DATA_NOT_AVAILABLE`	There aren't any buffers ready for the destination.
`B_DATA_AVAILABLE`	There are buffers ready for the destination.
`B_PRODUCER_STOPPED`	The producer has stopped.

RETURN CODES

- B_OK. The status update was sent without error.
- Port errors. The status update couldn't be delivered.

SetBufferGroup()

```
virtual status_t SetBufferGroup(const media_source &forSource,
      BBufferGroup *group) = 0
```

When a client wants a specific BBufferGroup to be used for a given output *forSource*, it will call this function. You should remember the *group* and use it for all requests for buffers to send on the output *forSource* (and for no other outputs, unless the client explicitly requests you do so by calling `SetBufferGroup()` for another output source).

If your BBufferProducer goes away, or the connection is broken, delete the BBufferGroup object.

If *group* is **`NULL`**, you should use whatever BBufferGroup you wish after disposing of the previous group.

It's okay to pass *group* on to another node upstream from your BBufferProducer if your BBufferProducer only passes along buffers it receives in its processing loop; in that case, you're not really the owner of the BBufferGroup, unless you pass **`true`** for *willReclaim* in the call to **`BMediaRoster::SetOutputBuffers()`**.

Return **`B_OK`** if the buffer group is set without incident; otherwise, return an appropriate error code.

SetInitialLatency()

```
void SetInitialLatency(bigtime_t initialLatency, uint32 flags) = 0
```

If your node has additional startup latency imposed by the signal from which its buffers are constructed, you should call `SetInitialLatency()` to specify the maximum possible latency that can be added by this delay. *initialLatency* should be the maximum latency, in microseconds, that might occur.

One situation in which this occurs is for TV capture card nodes. An NTSC television signal broadcasts a new field about every sixtieth of a second, which means that if your node is started partway through one field being received, you might have to wait as long as a sixtieth of a second for the first complete frame to arrive. So the maximum latency in this situation is a sixtieth of a second.

Setting the initial latency correctly can prevent consumers from having problems synchronizing with your node, and can improve performance.

flags should be 0 for now; there are no values defined yet.

SetPlayRate()

```

```

```
virtual status_t SetPlayRate(int32 numerator,
    int32 denominator)
```

This function is called to tell the producer to resample the data rate by the specified factor. Specifying a value of 1 (ie, *numerator*/*denominator* = 1) indicates that the data should be output at the same playback rate that it comes into the node at. The format of the data should be unchanged.

For example, if you're playing a sound at 48 kHz, and you receive a call to `SetPlayRate()` with a numerator of 2 and a demoninator of 1 (double speed), you should resample so that you move twice as fast through the source data while keeping the output rate constant. You might do this by doing a brute-force resample to 24 kHz (which would result in twice the data rate) or do time-compression (which would retain the pitch).

As another example, if you're playing video at 30 frames per second, and your `SetPlayRate()` function is called with a ratio of 1:2 specified (half speed), you should continue sending 30 frames per second, but you need to arrange for the playback to look like half-speed. A reasonable way to do this would be to send each frame twice (re-time-stamped and buffered internally, if necessary), which would result in the desired half-speed appearance.

Return **`B_OK`** if the sampling rate is changed; otherwise, return an error. It's okay to return an error if you don't support varying sampling rates—the Media Kit won't hold that against you.

VideoClippingChanged()

```
virtual status_t VideoClippingChanged(const media_source &forSource,
    int16 numShorts,
    int16 *clipData,
    const media_video_display_info &display,
    int32 *outFromChangeTag) = 0
```

This hook function is called when a client wants your BBufferProducer to output video data clipped to a particular region. Your producer must remember this clipping region and apply it to all video data you produce, without altering any bytes outside the region in any buffers sent through the source *forSource*.

Before your implementation of `VideoClippingChanged()` returns, you should set the value pointed to by *outFromChangeTag* to the change tag value at which the clipping will take effect, so the client will know what buffers it can expect to have the requested clipping. This can be done easily by adding the following line to your implementation:

```
*outFromChangeTag = UpdateChangeTag();
```

You can use the **`ClipDataToRegion()`** function to convert the data in *clipData* into an actual BRegion if that's a better format for you to work with. If you do, keep in mind that *numShorts* is the actual number of int16 values in the array specified by *clipData*, while **`ClipDataToRegion()`**

requires the number of **bytes** of data in the array; be sure to multiply *numShorts* by sizeof(int16).

The media_video_display_info structure referred to by display indicates the format of the video display onto which the video is being displayed; this lets you know what color space, screen size, and so forth is in use on the video display, so your producer can render properly. `VideoClippingChanged()` is called not only when clipping changes, but when the configuration of the display changes as well. Your producer must abide by this starting at the specified change count.

See "Video Clipping" on page87 for information on the format of the clip data.

See also: `ClipDataToRegion()`, `BBufferConsumer::RegionToClipData()`

Constants

Clipping Data Formats

Declared in: be/media/BufferProducer.h

Constant	Description
`B_CLIP_SHORT_RUNS`	Clipping is encoded using runs of shorts.

This value defines the only clipping format currently supported by BBufferProducer. Note that because this constant is a member of the BBufferProducer class, if you need to access it from other classes, you must code it as `BBufferProducer::B_CLIP_SHORT_RUNS`. See "Video Clipping" on page87 for a description of this format.

suggestion_quality

Declared in: be/media/BufferProducer.h

Constant	Meaning
`B_ANY_QUALITY`	Any quality.
`B_LOW_QUALITY`	A low quality level (10).
`B_MEDIUM_QUALITY`	Medium quality level (50).
`B_HIGH_QUALITY`	High quality level (100).

Quality values you can use when you don't want to have to come up with one on your own.

The Be Book The Media Kit The Media Kit Index

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferProducer Function Summary

Function Summary

Function	Description
AdditionalBufferRequested()	Hook: implement to immediately send the next buffer
BBufferProducer()	Constructor
ChangeFormat()	Tells the specified BBufferConsumer to accept a new format
ClipDataToRegion()	Converts data in the specified clipping format into a BRegion
Connect()	Hook: establish a connection between a source and a destination
Disconnect()	Hook: terminate a connection
DisposeOutputCookie()	Hook: called to dispose of an output iteration cookie
EnableOutput()	Hook: called to indicate whether or not a particular output needs to be send buffers
FindLatencyFor()	Returns the latency introduced by sending data to a specified destination
FindSeekTag()	Finds the tag of a

	previously-seen buffer to expedite seeking
FormatChangeRequested()	Hook: process a format change request
FormatProposal()	Hook: reports whether or not a suggested format is suitable for a given output
FormatSuggestionRequested()	Hook: returns a suggested media format
GetLatency()	Hook: reports the total latency incurred while processing a buffer
GetNextOutput()	Hook: returns information about available outputs
HandleMessage()	Dispatches a BBufferProducer message
LateNoticeReceived()	Hook: called when your latency is too high
LatencyChanged()	Hook: called a downstream consumer's latency changes
PrepareToConnect()	Hook: called to let the producer specialize wildcards and return a media_source to use for a connection
ProducerType()	Returns the type of media data the node produces
ProposeFormatChange()	Reports whether or not a given destination is ready to

	accept a specified format
SendBuffer()	Sends a buffer to a specified connected destination
SendDataStatus()	Informs a specified destination whether or not data is available
SetBufferGroup()	Hook: specifies that the BBufferProducer use a specific BBufferGroup
SetInitialLatency()	Sets the additional maximum startup latency imposed by the node
SetPlayRate()	Hook: specifies the playback rate for the output
VideoClippingChanged()	Hook: specifies the clipping region for a video output source

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferConsumer

Derived from: virtual BMediaNode
Declared in: be/media/BufferConsumer.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

BBufferConsumer is the counterpart to the BBufferProducer class—it receives BBuffers from the BBufferProducers that are connected to it, manipulates them in some fashion (either by altering the contents of the buffer or by playing the buffer's data to the speakers or to the screen), and possibly then passes them along to another BBufferConsumer (if the node also inherits from BBufferProducer).

The functions in this class aren't called by applications or by other nodes; they're called exclusively by the Media Kit to control and obtain information about a buffer consumer.

A BBufferConsumer publishes certain inputs, identified by `media_destination` structures, on which connections may be requested by a client application.

Dealing With Multiple Time Sources

Sometimes you'll find that the producer that's sending buffers to your BBufferConsumer is using a different time source than you are. It's relatively easy to cope with this situation—you just have to be aware that it can occur and compensate when it does:

```
bigtime_t time;
media_node producerNode;
BTimeSource *producerTimeSource;
media_node_id producerTSID = buffer->Header()->time_source;
if (producerTSID != myTSID) {
   roster->GetNodeFor(producerTSID, &producerNode);
   producerTimeSource = roster->MakeTimeSourceFor(&producerNode);
   time = buffer->Header()->start_time;
   time = producerTimeSource->RealTimeFor(time, myLatency);
   time = myTimeSource->PerformanceTimeFor(time);
}
else {
   time = buffer->Header()->start_time;
```

After this code has executed, *time* contains the start time for the buffer, in your node's time base.

You should probably, for performance's sake, cache the producer's time source the first time **BufferReceived()** is called.

Hook Functions

- AcceptFormat()
- BufferReceived()
- Connected()
- Disconnected()
- DisposeInputCookie()
- GetLatencyFor()
- GetNextInput()
- HandleMessage()
- ProducerDataStatus()
- FormatChanged()
- SeekTagRequested()

Constructor and Destructor

BBufferConsumer()

protected:

```
BBufferConsumer(media_type consumerType)
```

The BBufferConsumer constructor. Specify as *consumerType* the type of data the BBufferConsumer accepts.

In BeOS Release 4.5.2 and earlier *consumerType* has a default value. It no longer does. You'll have to actually specify the media type from now on.

~BBufferConsumer

protected:

```
~BBufferConsumer()
```

The BBufferConsumer destructor. You can augment this to handle whatever closing-out your consumer node requires.

If your node has created and set BBufferGroups for any producers, you should delete them in the destructor.

Member Functions

AcceptFormat()

```
virtual status_t AcceptFormat(const media_destination &destination,
    media_format *format) = 0
```

Implement this hook function to check that the specified *format* is reasonable for the specified destination, and to fill in any wildcard fields for which your BBufferConsumer has specific requirements.

If the format isn't reasonable (or is of a class that's unsuitable for *dest*), return `B_MEDIA_BAD_FORMAT`.

When `AcceptFormat()` returns `B_OK`, the Media Kit will expect a connection request on *destination* with the specified *format* not to fail due to a format incompatibility.

Don't try to ask the upstream producer about the format; it's waiting synchronously for your response, and doing so will cause deadlock.

BufferReceived()

protected:

```
virtual void BufferReceived(BBuffer *buffer) = 0
```

When a BBufferProducer sends buffers to one of your BBufferConsumer's inputs, it will eventually arrive here, at the `BufferReceived()` function (usually after first being dispatched by `HandleMessage()`).

Override this hook function to add the *buffer* to your internal playback queue, or to do whatever your node needs to do with buffers you consume. If you implement both BBufferProducer and BBufferConsumer, it's possible you might examine or alter the data in the *buffer* and then call BBufferPr`oducer::SendBuffer()` to send it along to someone else.

If your processing of the buffer is long enough to cause the buffer to become late, you should still process it as usual, but you should also call `NotifyLateProducer()` to let the producer know things are starting to lag.

Information about the contents and timing requirements of the *buffer* can be obtained by calling `BBuffer::Header()` on it.

If you're writing a node, and receive a buffer with the `B_SMALL_BUFFER` flag set, you must recycle the buffer before returning.

Connected()

protected:

```
virtual status_t Connected(const media_source &source,
    const media_destination &destination,
    const media_format *format,
    media_input *outInput) = 0
```

This hook function is called when a connection is being established to your input *destination* from the specified *source* producer. The connection will be composed of media data with the specified *format* (which you've previously accepted via **AcceptFormat()**).

Your implementation of `Connected()` should do whatever preparation you need to do to handle data input on the connection, and fill out the *outInput* buffer with information about the connection from your node's point-of-view. You can set *outInput*'s **destination** field different from *destination* if *destination* is a global connection-establishing input that's used to negotiate a connection, then create a new input to actually handle the data stream.

Since your BBufferConsumer has already had the opportunity to reject the specified *format*, it's poor form to return an error from this function. You should only return an error if the resources needed to establish the connection have become unavailable prior to the time `Connected()` was called.

On entry, *outInput*'s name field contains the name given the connection by the producer (this may be an empty string if the producer didn't assign a name). Your consumer should always make sure there's a valid name here, because it's a bad thing to have unnamed connections, and there's no guarantee that the producer will fill this in. If you don't have a good, descriptive name for a connection, the name should minimally contain the name of the node and a number that makes the connection's name unique (such as "MyNode Input 1" or "MyNode Output 3").

If you want the producer to use a specific BBufferGroup (for example, if you want a video producer to fill BDirectWindow buffers), you should create the BBufferGroup here, then call **BMediaRoster::SetOutputBuffersFor()** to set the producer's buffer group:

```
BBufferGroup *buffers = new BBufferGroup;
BMediaRoster::Roster()->SetOutputBuffersFor(producer, buffers);
```

Return **B_OK** if the connection is started safely, otherwise, return an appropriate error code.

ConsumerType()

```
media_type ConsumerType(const void)
```

Returns the type of media the BBufferConsumer consumes.

Disconnected()

protected:

```
virtual void Disconnected(const media_source &producer,
    const media_destination &whichInput) = 0
```

This hook function is called when a connection is being terminated. You should do whatever needs to be done in order to ensure that future inquiries about the media_source connected to the media_input indicated by *whichInput* reference **media_source::null** (or, if another connection is later established on the input, that producer).

If your consumer node has created and set a BBufferGroup for the producer, you shouldn't delete or reclaim it here, because the producer has a clone of the BBufferGroup that references the same buffers; deleting the BBufferGroup would free those buffers, leaving the producer in deadlock. Instead, delete (or reclaim) the when **Connected()** is called again, and be sure to delete any remaining BBufferGroups in your destructor.

DisposeInputCookie()

protected:

```
virtual void DisposeInputCookie(int32 cookie) = 0
```

If the cookie value you return in **GetNextInput()** is a pointer to an object that needs to be deleted when the iteration process is completed, be sure to implement `DisposeInputCookie()` to do so.

RequestFormatChange()

GetLatencyFor()

protected:

```
virtual status_t GetLatencyFor(const media_destination &forWhom,
    bigtime_t *outLatency,
    media_node_id *outTimeSource) = 0
```

Implement this hook function to calculate the total latency for the media_destination specified by *forWhom* and store the resulting value in *outLatency*. Also, return the time source your node is slaved to in *outTimeSource*.

If your node is a BBufferProducerRequestFormatChange() as well as a BBufferConsumer, you should include in your computed latency the latency of any nodes downstream from the specified connection that are slaved to the same time source.

Return **B_OK** if you successfully compute the latency; otherwise, return an appropriate error.

GetNextInput()

virtual status_t **GetNextInput(**int32 **cookie*, media_input **outInput***)** = 0

The first time a client calls this function, the value pointed to by *cookie* will be 0. You should fill the buffer pointed to by *outInput* with information about your first input, and set the value at *cookie* to something (other than zero) that will let you keep track of what to return the next time `GetNextInput()` is called.

Each successive call to `GetNextInput()` will pass back, in *cookie*, the value you returned in *cookie* the last time the function was called by that client, and you should fill *outInput* with information about the next input, and store a new value in *cookie* to continue to track your progress through the inputs.

Whenever this function is called with a value of zero in *cookie*, you must start over with the first input.

When you reach the last input, return **B_BAD_INDEX** to indicate that there aren't any more inputs.

HandleMessage()

protected:

virtual status_t **HandleMessage(**int32 *message*, const void **data*, size_t *size***)**

When your node derived from BBufferConsumer receives a message on its control port, you should try dispatching it by calling **BBufferConsumer::HandleMessage()**. If BBufferConsumer doesn't understand the message, it'll return **B_ERROR** and you can try dispatching it to another class from which your node is derived, or handle it yourself.

If this function returns **B_OK**, the message has been handled.

See also: **BMediaNode::HandleMessage()**, "About Multiple Virtual Inheritance" on page21

NotifyLateProducer()

static:

void **NotifyLateProducer(**const media_source &*source*, bigtime_t *howLate*,

bigtime_t *performanceTime*)

Notifies the BBufferProducer specified by *source* that it's running late by *howLate* microseconds; the notification conditions as of the specified *performanceTime*. Call this function when you detect that data is arriving too late and the run mode is **B_DECREASE_PRECISION**, **B_INCREASE_LATENCY**, or **B_DROP_DATA** (any of which permits adjustment of the media playback to maintain timeliness).

The producer should process this notification immediately and take the appropriate action.

ProducerDataStatus()

protected:

virtual void **ProducerDataStatus(**const media_destination &*destination*,
int32 *status*, bigtime_t *atPerformanceTime*) = 0

This hook function is called to inform your consumer about changes in the availability of buffers from the producer that's connected to the input *destination*. The *status* argument specifies what change has occurred, and *atPerformanceTime* indicates when the change happened (or when it will happen).

This lets you keep track of which inputs you should await data from; for example, if your consumer is processing data arriving from four producers, and one of them stops sending buffers to the consumer, the producer that's stopping will cause a call to `ProducerDataStatus()` to let you know not to await buffers anymore. This way, you know that when buffers have arrived from the other three inputs, it's okay to begin processing the buffers.

Constant	Meaning
B_DATA_NOT_AVAILABLE	The producer doesn't have any data available.
B_DATA_AVAILABLE	The producer has data available.
B_PRODUCER_STOPPED	The producer has been stopped.

RegionToClipData()

protected:

static status_t **RegionToClipData(**const BRegion **region*, int32 **format*,
int32 **ioSize*, void **data*)

Converts a BRegion into the clipping format used internally by the Media Kit. Prior to calling `RegionToClipData()`, *ioSize* is set to the size of the buffer pointed to by *data*. On return, *format* is the format of the clipping data, *ioSize* is changed to the actual number of bytes of data returned, and *data* contains the actual clipping data.

The clip data format is described in the section "Video Clipping" on page87.

RETURN CODES

- B_OK. Clip data returned without errors.
- **B_NO_MEMORY**. The *data* buffer isn't big enough.

See also: "Video Clipping" on page87

RequestAdditionalBuffer()

protected:

```
status_t RequestAdditionalBuffer(const media_source &source,
    BBuffer *previousBuffer,
    void *_reserved_ = NULL)
status_t RequestAdditionalBuffer(const media_source &source,
    bigtime_t startTime,
    void *_reserved_ = NULL)
```

Asks the upstream producer specified by *source* to immediately send the next buffer, instead of waiting until the appropriate time. The most obvious use for this function is in cases where a codec requires multiple buffers in order to decode a frame of output (MPEG is a good example).

The requested buffer can be identified either by a *startTime* parameter, which indicates the time for which a buffer is requested, or by a *previousBuffer*, which specifies the buffer prior to the one being requested.

This function will cause the producer's **AdditionalBufferRequested()** function to be called.

RETURN CODES

- B_OK. The change was requested successfully.
- **B_MEDIA_BAD_SOURCE**. The *source* isn't valid.
- **B_BAD_VALUE**. The *previousBuffer* pointer is **NULL**.
- **B_TIMEOUT**. The request to the Media Server timed out.
- Port errors. An error occurred communicating with the Media Server.

RequestFormatChange , FormatChanged()

protected:

```
status_t RequestFormatChange(const media_source &source,
    const media_destination &destination,
    media_format &toFormat,
```

```
    void *userData,
    int32 *changeTag,
    void *_reserved_ = NULL)
virtual status_t FormatChanged(const media_source &source,
    const media_destination &destination,
    int32 changeTag,
    media_format &newFormat) = 0
```

`RequestFormatChange()` requests that the producer *source* connected to the consumer *destination* change the format it produces to the format specified by *toFormat*. The Media Kit returns in *changeTag* the tag value that will be received by your **`RequestCompleted()`** function once the change takes effect; the change tag lets you match up the call to **`RequestCompleted()`** with this request. This function will receive a **`media_request_info`** structure with the indicated *userData* and *changeTag*.

`FormatChanged()` is called by the upstream producer when the media format your node will be receiving changes, and indicates the new format in *newFormat* and the change tag value at which the new format will take effect in *changeTag*. You should implement this function so your node will know that the data format is going to change. Note that this may be called in response to your **`AcceptFormat()`** call, if your **`AcceptFormat()`** call alters any wildcard fields in the specified format.

Because `FormatChanged()` is called by the producer, you don't need to (and shouldn't) ask it if the new format is acceptable.

If the format change isn't possible, return an appropriate error from `FormatChanged()`; this error will be passed back to the producer that initiated the new format negotiation in the first place.

RETURN CODES

- B_OK. No error.
- **`B_MEDIA_BAD_SOURCE`**. The specified source isn't valid.
- **`B_MEDIA_BAD_DESTINATION`**. The specified destination is invalid.
- Port errors. See Ports.

SeekTagRequested()

protected:

```
virtual status_t SeekTagRequested(const media_destination &destination,
    bigtime_t inTargetTime, uint32 inFlags,
    media_seek_tag *outSeekTag, bigtime_t *outTaggedTime,
    uint32 *outFlags)
```

This function is provided to aid in supporting media formats in which the outer encapsulation layer doesn't supply timing information. Producers will tag the buffers they generate with seek tags; these tags can be used to locate key frames in the media data.

It's the consumer's job to match up seek tags with performance times. As the consumer processes each incoming buffer, it should cache the seek tag and the performance time at which it occurs (if there's a tag on the buffer). When the producer needs to know the seek tag and corresponding time that's closest to a given performance time, this function is resonsible for returning that information.

RETURN CODES

- B_OK. No error.
- Other errors, depending on the node's implementation.

See also: "Seek Tags" on page12

SendLatencyChange()

protected:

```
status_t SendLatencyChange(const media_source &source,
    media_destination &destination,
    bigtime_t newLatency,
    uint32 flags = 0)
```

Lets the upstream producer know that the consumer node's latency has changed. *newLatency* indicates your new latency, in microseconds. The *flags* are currently unused and should always be 0.

You should call this whenever something happens to cause a change in your latency.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_SOURCE**. The source is invalid.
- **B_MEDIA_BAD_DESTINATION**. The destination is invalid.
- **B_TIMED_OUT**. The attempt to communicate with the Media Server timed out.
- Port errors. An error occurred communicating with the Media Server.

SetOutputBuffersFor()

protected:

```
status_t SetOutputBuffersFor(const media_source &source,
    const media_destination &destination,
    BBufferGroup *group,
    void *userData,
    int32 *changeTag,
    bool willReclaim = false,
    void *_reserved_ = NULL)
```

Specifies that the BBufferGroup *group* will provide the buffers for the connection between *source* and *destination*. If *willReclaim* is **false**, the Media Kit will dispose of the group for you; you can forget about it once this call returns. Otherwise, you're informing the Media Server that you want the group back, and that you'll delete it when you're done with it.

The Media Kit returns in *changeTag* the tag value that will be received by your `RequestCompleted()` function once the change takes effect; the change tag lets you match up the call to `RequestCompleted()` with this request. This function will receive a `media_request_info` structure with the indicated *userData* and *changeTag*.

The ability to request that certain buffers be used by a particular output can save you from having to perform unnecessary copies; you might be able to use BBuffers that represent a graphics card frame buffer, for example, so that a video producer's output goes directly to video memory.

Before reclaiming your buffers, be sure to call `SetOutputBuffersFor(output, NULL)` to let the Media Kit know your producer no longer has permission to use them. If you forget this step, the producer will hang onto the buffers until it's deleted, and your `BMediaRoster::ReclaimAllBuffers()` call will hang, possibly forever.

RETURN CODES

- B_OK. The change was requested successfully.
- Other errors. The change could not be made.

SetOutputEnabled()

protected:

```
static status_t SetOutputEnabled(const media_source &source,
    const media_destination &destination,
    bool enabled,
    void *userData,
    int32 *changeTag,
    void *_reserved_ = NULL)
```

Specifies whether or not the specified *output* should be transmitting buffers to the *destination*. If *enabled* is **true**, the producer should transmit buffers; otherwise it should not.

The Media Kit returns in *changeTag* the tag value that will be received by your `RequestCompleted()` function once the change takes effect; the change tag lets you match up the call to `RequestCompleted()` with this request. This function will receive a `media_request_info` structure with the indicated *userData* and *changeTag*.

RETURN CODES

- B_OK. The change was requested successfully.
- `B_MEDIA_BAD_SOURCE`. The *source* is invalid.

SetVideoClippingFor()

protected:

```
status_t SetVideoClippingFor(const media_source &output,
    const media_destination &destination,
    const int16 *shortsList,
    int32 shortCount,
    const media_video_display_info &display,
    void *userData,
    int32 *changeTag,
    void *_reserved_ = NULL)
```

This function requests that video buffers sent by the specified *output* to the specified *destination* clip all its writing in buffers it sends to the BBufferConsumer to the clipping region described by *shortsList* and *shortCount*. The clip data format is described in the section "Video Clipping" on page87.

The Media Kit returns in *changeTag* the tag value that will be received by your `RequestCompleted()` function once the change takes effect; the change tag lets you match up the call to `RequestCompleted()` with this request. This function will receive a `media_request_info` structure with the indicated *userData* and *changeTag*.

The `media_video_display_info` structure referenced by *display* describes the current configuration of the video in terms of color space, resolution, and so forth.

RETURN CODES

- B_OK. The clipping request has been sent without errors.
- `B_MEDIA_BAD_CLIP_FORMAT`. The clipping data isn't formatted correctly.
- Port errors. See Ports.

See also: "Video Clipping" on page87

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferConsumer Function Summary

Function Summary

Function	Description
AcceptFormat()	Hook: lets you determine whether or not a suggested output format is reasonable
BBufferConsumer()	Constructor
BufferReceived()	Hook: called when a buffer arrives
Connected()	Hook: called when a connection is being established
ConsumerType()	Returns the media type the BBufferConsumer consumes
Disconnected()	Hook: called when a connection is being terminated
DisposeInputCookie()	Hook: dispose of the cookie used for iterating over inputs
FormatChanged()	Hook: called when the media format changes
GetLatencyFor()	Hook: return the total latency for the specified destination
GetNextInput()	Hook: returns the next input iteratively
HandleMessage()	Dispatches a BBufferConsumer message
NotifyLateProducer()	Notify the specified producer that it's running late, and by how

	much
ProducerDataStatus()	Hook: provides information about availability changes from a given producer
RegionToClipData()	Given a BRegion, returns clipping information suitable for use with BBufferConsumer clipping functions
RequestAdditionalBuffer()	Asks the producer to send a buffer immediately instead of waiting until it's time to be sent
SeekTagRequested()	Asks the producer to return a seek tag for a given media time
SendLatencyChange()	Asks the producer to return a seek tag for a given media time
SetOutputBuffersFor()	Sets the BBufferGroup used to provide buffers for a particular connection
SetOutputEnabled()	Specify whether or not a given media_source should be outputting buffers
SetVideoClippingFor()	Request that the specified output clip to a given region

---

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBuffer

Derived from: none
Declared in: be/media/Buffer.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A BBuffer references a chunk of shared memory where media data can be passed between applications and nodes. The control header for these buffers is passed through the use of a safer IPC mechanism, but the actual data these headers represent typically require a high data rate, and for the sake of maximizing performance, shared memory is the method by which the buffers are managed.

BBuffers originate at some BBufferProducer, which has a BBufferGroup that acts as a source of BBuffers (see `BBufferConsumer::SetOutputBuffersFor()`). BBuffers are sent through all participating nodes, possibly being sent down a long chain—much like a bucket brigade—until finally the node arrives at a node that chooses not to pass along the BBuffer any further, at which the buffer's `Recycle()` function is called to return the buffer to the BBufferGroup's store, where it gets reused by another batch of data.

An application or custom node can set up BBuffers that reference a specific area of memory, such as a low-level driver buffer, a BBitmap, or a BDirectWindow's area of a frame buffer. This ability can provide for great optimization by avoiding unnecessary copy operations.

The BSmallBuffer class, derived from BBuffer, is used for very small buffers; they don't go in BBufferGroups, and get handled specially to optimize performance.

BBuffer should never be subclassed, since they're mostly owned and managed by the Media Server. Even if you create your own BBuffer, once you've called `BBufferConsumer::SetOutputBuffersFor()`, the "same" buffer received back by your BBufferConsumer may actually be another instance of the BBuffer class.

Constructor and Destructor

The BBuffer constructor and destructor are private; use the appropriate functions in the BBufferGroup class to create and destroy BBuffers.

Member Functions

AudioHeader()

```
media_audio_header *AudioHeader(void)
```

Returns a pointer to an audio buffer's header; this is just an alias for:

```
&BBuffer::Header()->u.raw_audio;
```

CloneInfo()

```
buffer_clone_info CloneInfo(void) const
```

Returns a buffer_clone_info structure describing a buffer. This information is primarily for debugging purposes; don't use this buffer_clone_info structure to add a buffer to a buffer group, because it'll just alias an existing buffer.

Data()

```
void *Data(void)
```

The `Data()` function returns a pointer to the first byte of the buffer, or **`NULL`** if the buffer somehow couldn't be properly initialized (in which case the buffer can't be used).

Flags()

```
int32 Flags(void)
```

Returns the buffer's flags. The flags let you specify options for the buffer. For example, you might create a BBufferGroup in which some buffers are intended for odd video fields (**`B_F1_BUFFER`**), and other buffers are intended for even video fields (**`B_F2_BUFFER`**), thereby letting you handle interlaced video much more easily by having each buffer know where the video should be displayed.

Header()

```
media_header *Header(void)
```

Returns a pointer to the buffer's header. This header describes the media data contained therein. The result is only valid for a buffer received from either **`BBufferGroup::RequestBuffer()`** or **`BBufferConsumer::BufferReceived()`**.

If you put data into a buffer, you should call `Header()` to obtain a pointer to the buffer's header, then fill out the media_header structure with information describing the buffer's contents.

ID()

media_buffer_id **`ID(`**void**`)`**

If the buffer has been successfully registered with the Media Server, and is available for use by other clients, `ID()` returns a positive buffer ID. Otherwise, a negative number is returned.

Recycle()

void **`Recycle(`**void**`)`**

Sends the buffer back to the BBufferGroup that owns it so the buffer can be reused. You can only call `Recycle()` on a buffer that you received from either the **`BBufferGroup::RequestBuffer()`** call or the **`BBufferConsumer::BufferReceived()`** call.

Don't call both **`BroadcastBuffer()`** and `Recycle()` on the same buffer.

SizeAvailable()

size_t **`SizeAvailable(`**void**`)`**

Returns the size, in bytes, of the buffer. The actual number of bytes used might be less than this value, and is stored in the buffer's header, which can be obtained by calling the **`Header()`** function.

SizeUsed() , SetSizeUsed()

size_t **`SizeUsed(`**void**`)`**

void **`SetSizeUsed(`**ssize_t *sizeUsed*`)`**

`SizeUsed()` returns the number of bytes in the buffer that are currently in use.

`SetSizeUsed()` sets this value. This should be called after writing data into the buffer in order to indicate the size of the written data.

Type()

```
media_type Type(void)
```

Returns the media type of the data contained within the buffer, as specified by the BBufferProducer from which the buffer originated. This value is only valid if you received the buffer from either the **BBufferGroup::RequestBuffer()** call or the **BBufferConsumer::BufferReceived()** call.

VideoHeader()

```
media_video_header *VideoHeader(void)
```

Returns a pointer to a video buffer's header; this is just an alias for:

```
&BBuffer::Header()->u.raw_video;
```

Constants

Flags

Declared in: be/media/Buffer.h

Constant	Description
`B_F1_BUFFER`	Buffer is for odd fields.
`B_F2_BUFFER`	Buffer is for even fields.
`B_SMALL_BUFFER`	The buffer is a small buffer.

These flags can be assigned to a buffer; the current possible values let you specify whether the buffer should be used for even video fields or odd video fields; if `B_SMALL_BUFFER` is set, the buffer is a BSmallBuffer.

Structures

buffer_clone_info

Declared in: be/media/Buffer.h

```
struct buffer_clone_info {
   buffer_clone_info();
   ~buffer_clone_info();
   media_buffer_id buffer;
   area_id area;
   size_t offset;
   size_t size;
   int32 flags;
private:
   _reserved_[4];
};
```

Describes where in memory a BBuffer resides (in terms of the memory area and offset into the area at which the buffer is located, and the size of the buffer), as well as the buffer's flags.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBuffer Function Summary

Function Summary

Function	Description
AudioHeader()	Returns a pointer to an audio buffer's header
CloneInfo()	Returns information about a buffer
Data()	Returns a pointer to the first byte of the buffer
Flags()	Returns the buffer's flags
Header()	Returns a pointer to the buffer's header
ID()	Returns the BBuffer's buffer ID
Recycle()	Sends the buffer back to the BBufferGroup it belongs to for reuse
SetSizeUsed()	Sets the number of bytes currently used in the buffer
SizeAvailable()	Returns the number of bytes the buffer can hold
SizeUsed()	Returns the number of bytes currently used in the buffer
Type()	Returns the type of media within the buffer
VideoHeader()	Returns a pointer to a video buffer's header

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferGroup

Derived from: none
Declared in: be/media/BufferGroup.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A BBufferGroup serves as a repository for BBuffers, and can be used to pass those buffers to another node for its use when sending data. It must also be used by your BBufferProducer to acquire BBuffers to put data that you produce into before you send the buffer on to its destination.

You can create a new BBufferGroup by simply calling:

```
new BBufferGroup;
```

If you pass in some arguments describing what sort of buffers you want, and how many of them, the group will even create the BBuffers for you. This is how you'll usually use BBufferGroup, from within your BBufferProducer subclass' constructor so you'll know there are buffers ready and waiting when it's time to start sending data. A possible exception is if your node is a filter that simply processes buffers it receives and passes them along.

A BBufferGroup instance runs a thread that reclaims BBuffers whose `Recycle()` function has been called. If the group is temporarily out of free buffers, a request for a buffer may block until one is available, or until the request times out, if a timeout is specified when the request is made.

Using BBitmaps as Buffers

If you're doing video processing, you might want your buffers to be BBitmaps. Here's how you can accomplish this:

```
BBufferGroup *my_group = new BBufferGroup;
BBitmap *my_bitmap = new BBitmap(BRect(0,0,639,479), B_BITMAP_IS_AREA,
            B_RGB32, 640*4);

if (!my_bitmap->IsValid()) {
   return B_ERROR;
}
area_info bm_info;

if (get_area_info(my_bitmap->Area(), &bm_info) != B_OK) {
   return B_ERROR;
}

buffer_clone_info bc_info;
bc_info.area = bm_info.area;
bc_info.offset = ((char *) my_bitmap->Bits())-((char *) bm_info.address);
bc_info.size = my_bitmap->BitsLength();

BBuffer *out_buffer = NULL;
status_t err = my_group->AddBuffer(bc_info, &out_buffer);

/* out_buffer is now set to use the BBitmap's memory */
```

This code takes advantage of the **`B_BITMAP_IS_AREA`** flag when creating the bitmap. This forces the bitmap to be in a memory area of its very own, which you can then use when creating the BBuffer.

If you need to do real-time accesses to the BBuffer, you should lock it down. In addition, if you use DMA, you'll need to specify for the buffer to be contiguous (you can do this using the BBitmap constructor, specifying the contiguous flag).

Before deleting any BBitmaps used in this way, be sure the buffer group has been deleted first and (if **`SetOutputBuffersFor()`** has been used) all buffers have been reclaimed.

Constructor and Destructor

BBufferGroup()

```
BBufferGroup(size_t size, int32 numBuffers = 3,
    uint32 placement = B_ANY_ADDRESS,
    uint32 lock = B_FULL_LOCK)
BBufferGroup(void)
BBufferGroup(int32 numBuffers, const buffer_id *bufferList)
```

The first form of the constructor creates a BBufferGroup with some number of BBuffers already allocated by the group. These buffers will all live within a single Kernel Kit area, allocated by the group. The group tries to allocate an area with properties specified by the *placement* and *lock* arguments, large enough to hold *numBuffers* buffers of the specified *size* (there may be some padding added).

The second form of the constructor creates a BBufferGroup but doesn't create any buffers for it. You should add BBuffers to the group before trying to use it.

The third form of the constructor creates a BBufferGroup that contains the specified list of buffers. *bufferList* points to an array of buffer IDs for the buffers to be controlled by the group, and *numBuffers* is the number of buffers in the list. This version of the constructor isn't one you'll use very often.

Before using any buffers from a new BBufferGroup, call `InitCheck()` to determine if any errors occurred while creating the group.

~BBufferGroup

```
~BBufferGroup()
```

Releases all memory used by the BBufferGroup, including the BBuffers it controls.

Keep in mind that `AddBuffer()` clones the buffer area. This destructor releases the clones, but it's your application's job to release the original area you added.

Your node needs to delete the buffer group corresponding to any connection that is disconnected or whenever the node is deleted. Until your node does so, the other end of the connection will be blocked waiting for the buffer group to be deleted.

Member Functions

AddBuffer()

```
status_t AddBuffer(const buffer_clone_info &info)
```

Given the buffer_clone_info, this function clones the buffer and adds the clone to the BBufferGroup. Normally you'll fill out the buffer_clone_info structure yourself.

Since the buffer is cloned, you'll need to delete the original memory area specified by the buffer_clone_info structure yourself when you're no longer using it; the BBufferGroup won't do it for you.

You shouldn't pass the result of a `BBuffer::CloneInfo()` call to this function, as doing so would create an "alias" buffer for the same memory area. This is probably not the effect you want.

RETURN CODES

- B_OK. No error adding the buffer to the group.
- `B_NO_MEMORY`. Couldn't clone the buffer into the BBufferGroup's address space.

AddBuffersTo()

```
status_t AddBuffersTo(BMessage *message, const char *name,
    bool needLock = true)
```

Adds the group's buffers to the specified *message*, storing them in an array with the specified *name*. If *needLock* is **true**, the BBufferGroup is locked before performing the operation, then unlocked when finished; otherwise, you guarantee that the buffers won't go anywhere during the `AddBuffersTo()` call.

The buffers are added by ID number, and are therefore in int32 format.

RETURN CODES

- B_OK. No error adding the buffers to the message.
- Errors from `AddInt32()`.

CountBuffers()

```
status_t CountBuffers(int32 *outBufferCount)
```

Returns, in *outBufferCount*, the number of buffers in the group.

RETURN CODES

- B_OK. The number of buffers was returned successfully.
- **B_BAD_VALUE**. Invalid outBufferCount pointer.

GetBufferList()

status_t **GetBufferList(**int32 *listCount*, BBuffer ***outBuffers***)**

Returns a list of all the buffers in the group. When calling `GetBufferList()`, pass in *outBuffers* a pointer to an array of BBuffer pointers that you want to be filled with pointers to the group's buffers, and specify the number of elements in the array in *listCount*.

RETURN CODES

- B_OK. No errors.
- **B_BAD_VALUE**. *listCount* is less than 1, or *outBuffers* is **NULL**

InitCheck()

status_t **InitCheck(**void**)**

Returns the error code resulting from the construction of the BBufferGroup.

RETURN CODES

- B_OK. The new group was created successfully.
- **B_ERROR**. Couldn't allocate the buffers for the group.
- Area errors. See Areas in The Kernel Kit.

ReclaimAllBuffers()

status_t **ReclaimAllBuffers(**void**)**

If you pass a buffer group to some other BBufferProducer but pass **true** for *willReclaim* in the **BMediaRoster::SetOutputBuffers()** call, you can later reclaim the buffers into the BBufferGroup by calling **Reclaim()**.

`ReclaimAllBuffers()` will return **B_OK** when all buffers are accounted for, or return an error if buffers can't be reclaimed.

If you have buffers that reference some object that might go away (such as a BBitmap), you should call `ReclaimAllBuffers()` on the group and delete the BBufferGroup before that object goes away.

Before reclaiming your buffers, be sure to call `BMediaRoster::SetOutputBuffersFor(output, NULL)` to let the Media Kit know your producer no longer has permission to use them. If you forget this step, the producer will hang onto the buffers until it's deleted, and your `ReclaimAllBuffers()` call will hang, possibly forever.

RETURN CODES

- B_OK. All buffers reclaimed successfully.
- **`B_MEDIA_CANNOT_RECLAIM_BUFFERS`**. Some buffers couldn't be reclaimed.

RequestBuffer()

BBuffer ***RequestBuffer(**size_t *size*, bigtime_t *timeout* = B_INFINITE_TIMEOUT**)**
status_t **RequestBuffer(**BBuffer ***outBuffer*, bigtime_t *timeout* = B_INFINITE_TIMEOUT**)**

Returns a pointer to a BBuffer of at least the specified *size* that your BBufferProducer subclass can put data into, then pass on to a **`media_destination`**. If there isn't a suitable buffer available, the call will block until either a buffer becomes available (in which case the buffer is returned) or the specified *timeout* is reached.

If you pass a *timeout* value that's less than zero, **`RequestBuffer()`** will return **`NULL`** immediately if there's no buffer available, otherwise it will return a pointer to a buffer you can use.

In BeOS Release 4.5, the *timeout* is ignored (unless you specify a negative value); **`B_INFINITE_TIMEOUT`** is always used, regardless of the value you specify.

`RequestBuffer()` doesn't use the buffer flags; instead, you can look at the buffers within a BBufferGroup to find a specific buffer to request yourself, based on the value returned by **`BBuffer::Flags()`** or **`BBuffer::Size()`**, for example. Use the second form of **`RequestBuffer()`** to obtain a specific buffer.

The first version of `RequestBuffer()` returns **`NULL`** if no buffer can be obtained, and sets errno to the appropriate error code. The second version returns an appropriate error code.

Don't call `RequestBuffer()` while an outstanding **`ReclaimAllBuffers()`** request is pending. To make sure that doesn't happen, a buffer producer should be designed to not know about the old group anymore once **`SetBufferGroup()`** is called to change its buffer group.

RETURN CODES

- B_OK. No errors.
- **B_MEDIA_BUFFERS_NOT_RECLAIMED**. Buffers are in the process of being reclaimed.
- **B_ERROR**. A miscellaneous error occurred.

RequestError()

```
status_t RequestError(void)
```

Returns the last **RequestBuffer()** error. This is useful if **RequestBuffer()** returns **NULL**.

WaitForBuffers()

```
status_t WaitForBuffers(void)
```

Waits until the currently pending buffer reclamation is finished, then returns. If there isn't a buffer reclamation in progress, returns immediately.

RETURN CODES

- B_OK. No error.
- Semaphore errors. Unable to acquire the semaphore used to detect that buffer reclamation is done.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBufferGroup Function Summary

Function Summary

Function	Description
AddBuffer()	Adds a BBuffer to the group
AddBuffersTo()	Adds the group's buffers to a BMessage
BBufferGroup()	Constructor
CountBuffers()	Returns the number of buffers in the group
GetBufferList()	Returns a list of all the buffers in the group
InitCheck()	Returns the error code resulting from the construction of the object
ReclaimAllBuffers()	Returns control of buffers to the BBufferGroup
RequestBuffer()	Returns a pointer to a buffer you can use
RequestError()	Returns the last RequestBuffer() error
WaitForBuffers()	Waits until buffer reclamation is complete

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBitmap

Derived from: BArchivable
Declared in: be/interface/Bitmap.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BBitmap describes a rectangular image as a two-dimensional array of pixel data (or *bitmap*). The BBitmap class lets you create a bitmap by specifying raw pixel data (through the `Bits()` and `SetBits()` functions), or you can add a BView to your BBitmap and use the view's drawing operations (`FillRect()`, `StrokeLine()`, etc) to draw into the BBitmap object (see "Using a View to Draw into a Bitmap", below).

The BBitmap class doesn't provide a way to actually display bitmap data. Displaying a bitmap is the task of BView functions such as `DrawBitmap()`.

Bitmap Data

A bitmap records the color values of every pixel within a rectangular area. The data is specified in rows, beginning with the top row of pixels in the image and working downward to the bottom row. Each row is aligned on a long word boundary.

When you construct a BBitmap object, you give it a bounds rectangle (integer coordinates only!) and a color space. For example, this code

```
BBitmap *image = new BBitmap(BRect(0.0, 0.0, 79.0, 39.0), B_CMAP8);
```

constructs a 40 x 80 bitmap of 8-bit color data.

The data in a BBitmap object isn't initialized—a BBitmap has no default background color. When you set the bitmap's data (however you set it) you must "paint" every pixel within the bitmap's boudns rectangle.

Using a View to Draw into a Bitmap

If you're going to use a view to draw into your bitmap, you must tell the BBitmap constructor by setting the third argument to true:

```
BBitmap *image = new BBitmap(BRect(...), B_CMAP8, true);
```

You then add the view to the bitmap (you don't have to do anything special when constructing the BView):

```
bitmap->AddChild(view);
```

When the view draws, the drawing operations are rendered into the bitmap. Note that you must explicitly tell the BView to draw—the BViews that you use to draw into a BBitmap aren't part of the user interface, so they won't receive user event messages. When you're done drawing, you should call BBitmap's **`Sync()`** function to make sure the drawing has all been performed. If the bitmap that you've created is static—if it doesn't need to change after you've drawn into it—you can throw away the BView that you used create the bitmap data.

A BBitmap can contain more than one BView—it can act as the root of an entire view hierarchy. The BBitmap class defines a number of BWindow-like functions—**`AddChild()`**, **`FindView()`**, **`ChildAt()`**, and so on—to help you create and manage the hierarchy.

Transparency

Color bitmaps can have transparent pixels. When the bitmap is imaged in a drawing mode other than **`B_OP_COPY`**, its transparent pixels won't be transferred to the destination view. The destination image will show through wherever the bitmap is transparent.

To introduce transparency into a **`B_CMAP8`** bitmap, a pixel can be assigned one of the following values, as appropriate for the bitmap's color space.

`B_TRANSPARENT_MAGIC_CMAP8`	8-bit indexed color transparent pixel.
`B_TRANSPARENT_MAGIC_RGBA15`	15-bit transparent pixel.
`B_TRANSPARENT_MAGIC_RGBA15_BIG`	15-bit transparent pixel, big-endian.
`B_TRANSPARENT_MAGIC_RGBA32`	32-bit transparent pixel.
`B_TRANSPARENT_MAGIC_RGBA32_BIG`	32-bit transparent pixel, big-endian.

Opaque pixels should have an alpha value of 255 for 8-bit alpha channels or 1 for 1-bit alpha channels; values of 0 indicate 100% transparent pixels. Values in between (for 8-bit alpha channels) represent varying degrees of transparency.

Transparency is covered in more detail under "Drawing Modes" on page26.

See also: **`system_colors()`**

Constructor and Destructor

BBitmap()

```
BBitmap(BRect bounds, color_space space,
    bool acceptsViews = false,
    bool needsContiguousMemory = false)
BBitmap(BMessage *archive)
```

The BBitmap class insists that a BApplication object be present (but not necessarily running).

Creates a new BBitmap object that can hold a bitmap whose size and depth are described by *bounds* and *space*. The bitmap data is uninitialized; you set the data through **Bits()**/**SetBits()**, or by drawing into an attached BView (see "Using a View to Draw into a Bitmap").

If BViews are to be used, the *acceptsViews* argument must be set to **true**. Furthermore (in this case), the origin of the *bounds* rectangle must be 0.0.

If the *needsContiguousMemory* flag is `true`, the BBitmap will make sure that the (physical) memory it allocates is one contiguous physical chunk. This should matter only to drivers doing direct DMA into physical memory.

The possible color spaces are enumerated in the section above titled "The Color Space".

~BBitmap()

```
virtual ~BBitmap()
```

Frees all memory allocated to hold image data, deletes any BViews used to create the image, gets rid of the off-screen window that held the views, and severs the BBitmap's connection to the Application Server.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BBitmap object—or `NULL`, if the *archive* message doesn't contain data for a BBitmap object. The new object is allocated by `new` and created with the version of the constructor that takes a BMessage *archive*.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

AddChild()

virtual void **AddChild(**BView **aView***)**

Adds *aView* (and all its children) to this BBitmap's view hierarchy, and causes `AttachedToWindow()` to be sent to the newly add children.

If *aView* already has a parent, the application may crash. Be sure to remove the view from a previous parent before trying to add it to a bitmap.

`AddChild()` fails if the BBitmap was not constructed to accept views.

See also: `BWindow::AddChild()`, `BView::AttachedToWindow()`, `RemoveChild()`, the BBitmap constructor

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of `Archive()` and stores the BBitmap in the BMessage *archive*.

See also: `BArchivable::Archive()`, `Instantiate()` static function

Bits()

void ***Bits(**void**)** const

Returns a pointer to the bitmap data. The length of the data can be obtained by calling `BitsLength()`—or it can be calculated from the height of the bitmap (the number of rows) and `BytesPerRow()`.

The data is in the format specified by `ColorSpace()`.

This pointer is valid throughout the entire lifespan of the object.

See also: `Bounds()`, `BytesPerRow()`, `BitsLength()`

BitsLength()

int32 **BitsLength(**void**)** const

Returns the number of bytes that were allocated to store the bitmap data.

See also: `Bits()`, `BytesPerRow()`

Bounds()

```
BRect Bounds(void) const
```

Returns the bounds rectangle that defines the size and coordinate system of the bitmap. This should be identical to the rectangle used in constructing the object.

BytesPerRow()

```
int32 BytesPerRow(void) const
```

Returns how many bytes of data are required to specify a row of pixels. This may include slop space required by the graphics hardware; you should always use this call to determine the width of a row of pixels in bytes instead of assuming that it will be the number of pixels multiplied by the size of a pixel in bytes.

ChildAt() , CountChildren()

```
BView *ChildAt(int32 index) const
int32 CountChildren(void) const
```

`ChildAt()` returns the child BView at *index*, or `NULL` if there's no child at *index*. Indices begin at 0 and count only BViews that were added to the BBitmap (added as children of the top view of the BBitmap's off-screen window) and not subsequently removed.

`CountChildren()` returns the number of BViews the BBitmap currently has. (It counts only BViews that were added directly to the BBitmap, not BViews farther down the view hierarchy.)

These functions fail if the BBitmap wasn't constructed to accept views.

ColorSpace()

```
color_space ColorSpace(void) const
```

Returns the color space of the data being stored (not necessarily the color space of the data passed to the `SetBits()` function). Once set by the BBitmap constructor, the color space doesn't change.

CountChildren() see ChildAt()

FindView()

BView ***FindView**(BPoint *point*)* const
BView ***FindView**(*const char *name*)* const

Returns the BView located at *point* within the bitmap or the BView tagged with *name*. The point must be somewhere within the BBitmap's bounds rectangle, which must have the coordinate origin, (0.0, 0.0), at its left top corner.

If the BBitmap doesn't accept views, this function fails. If no view draws at the *point* given, or no view associated with the BBitmap has the *name* given, it returns `NULL`.

IsValid()

bool **IsValid(**void**)** const

Returns `true` if there's memory for the bitmap (if the address returned by **`Bits()`** is valid), and `false` if not.

Lock() , Unlock() , IsLocked()

bool **Lock(**void**)**

void **Unlock(**void**)**

bool **IsLocked(**void**)** const

These functions lock and unlock the off-screen window where BViews associated with the BBitmap draw. Locking works for this window and its views just as it does for ordinary on-screen windows.

`Lock()` returns `false` if the BBitmap doesn't accept views or if its off-screen window is unlockable (and therefore unusable) for some reason. Otherwise, it doesn't return until it has the window locked and can return `true`.

`IsLocked()` returns `false` if the BBitmap doesn't accept views. Otherwise, it returns the lock status of its off-screen window.

RemoveChild()

virtual bool **RemoveChild(**BView **aView***)**

Removes *aView* from the hierarchy of views associated with the BBitmap, but only if *aView* was added to the hierarchy by calling BBitmap's version of the **`AddChild()`** function.

If *aView* is successfully removed, **`RemoveChild()`** returns `true`. If not, it returns `false`.

SetBits()

void **SetBits(**const void **data*, int32 *length*, int32 *offset*, color_space *mode***)**

Assigns *length* bytes of *data* to the BBitmap object. The new data is copied into the bitmap beginning *offset* bytes (*not* pixels) from the start of allocated memory. To set data beginning with the first (left top) pixel in the image, the *offset* should be 0; to set data beginning with, for example, the sixth pixel in the first row of a `B_RGB32` image, the offset should be 20. The offset counts any padding required to align rows of data.

This function is intended to be used for importing existing data from a different format rather than for setting individual pixels in the bitmap. If you're interested in coloring individual pixels, use `Bits()` to obtain direct access to the bitmap data.

The source data is specified in the *mode* color space, which may or may not be the same as the color space that the BBitmap uses to store the data. If not, the following conversions are automatically made:

- `B_GRAY1` and `B_RGB32` to `B_CMAP8`.
- `B_CMAP8` and `B_GRAY1` to `B_RGB32`.

These are the *only* color conversions SetBits() understands; all other conversions must be performed manually.

Colors may be dithered in a conversion to `B_CMAP8` so that the resulting image will match the original as closely as possible, despite the lost information.

If the color space *mode* is `B_RGB32`, the *data* should be triplets of three 8-bit components—red, green, and blue, in that order—without an alpha component. Although stored as 32-bit quantities with the components in BGRA order, the input data is only 24 bits in RGB order. Rows of source data do not need to be aligned.

However, if the source data is in any *mode* other than `B_RGB32`, padding must be added so that each row is aligned on a `int32` word boundary.

`SetBits()` works only on BBitmaps in B_GRAY1, B_CMAP8, and B_RGB32 color spaces; all other conversions must be carried out manually.

This function works for all BBitmaps, whether or not BViews are also enlisted to produce the image.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_frame"	`B_RECT_TYPE`	The BBitmap's bounds rectangle.
"_cspace"	`B_INT32_TYPE`	The color_space of the data.
"_view_ok"	`B_BOOL_TYPE`	Always `true`, indicating the BBitmap accepts views (only present in deep copy archives of BBitmaps accepting views).
"_data"	`B_RAW_TYPE`	The bitmap data (present only if "_view_ok" not present).
"_continguous"	`B_BOOL_TYPE`	Whether the BBitmap requires memory in one contiguous chunk.

If the "_view_ok" field is present, the child views of the BBitmap are additionally archived in the "_views" array of BMessages. See the description of the BView Archived Fields for more information on those fields.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Interface Kit Table of Contents

- The Interface Kit
- Drawing
- Listening to the User
- The Coordinate Space
- BAlert
- BBitmap
- BBox
- BButton
- BCheckBox
- BColorControl
- BControl
- BDragger
- BFont
- BListItem
- BListItem: Creating a Custom List Item
- BListView
- BMenu
- BMenuBar
- BMenuField
- BMenuItem
- BOutlineListView
- BPicture
- BPictureButton
- BPoint
- BPolygon
- BPopUpMenu

- BPrintJob
- BRadioButton
- BRect
- BRegion
- BScreen
- BScrollBar
- BScrollView
- BShape
- BShapeIterator
- BShelf
- BSlider
- BSlider: Customizing your Slider
- BStatusBar
- BStringItem
- BStringView
- BTab
- BTabView
- BTextControl
- BTextView
- BView
- BView Introduction
- BView General Functions
- BView Hook Functions
- BView Input-Related Functions
- BView Graphics State Functions
- BView Drawing-Related Functions
- Scripting and Archival
- BView View Hierarchy Functions
- BWindow
- Global Functions
- Global Variables, Constants, and Defined Types

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Interface Kit: Master Index

!

!=	BFont
!=	BPoint
!=	BRect

&

&	BRect

+

+	BPoint
+=	BPoint

1

B_15_BIT_1024x768	Global Variables, Constants, and Defined Types
B_15_BIT_1152x900	Global Variables, Constants, and Defined Types
B_15_BIT_1280x1024	Global Variables, Constants, and Defined Types
B_15_BIT_1600x1200	Global Variables,

	Constants, and Defined Types
B_15_BIT_640x480	Global Variables, Constants, and Defined Types
B_15_BIT_800x600	Global Variables, Constants, and Defined Types
B_16_BIT_1024x768	Global Variables, Constants, and Defined Types
B_16_BIT_1152x900	Global Variables, Constants, and Defined Types
B_16_BIT_1280x1024	Global Variables, Constants, and Defined Types
B_16_BIT_1600x1200	Global Variables, Constants, and Defined Types
B_16_BIT_640x480	Global Variables, Constants, and Defined Types
B_16_BIT_800x600	Global Variables, Constants, and Defined Types

---

3

B_32_BIT_1024x768	Global Variables, Constants, and Defined Types
B_32_BIT_1152x900	Global Variables, Constants, and Defined Types
B_32_BIT_1280x1024	Global Variables, Constants, and Defined Types
B_32_BIT_1600x1200	Global Variables, Constants, and Defined Types

B_32_BIT_640x480	Global Variables, Constants, and Defined Types
B_32_BIT_800x600	Global Variables, Constants, and Defined Types

---

8

B_8_BIT_1024x768	Global Variables, Constants, and Defined Types
B_8_BIT_1152x900	Global Variables, Constants, and Defined Types
B_8_BIT_1280x1024	Global Variables, Constants, and Defined Types
B_8_BIT_1600x1200	Global Variables, Constants, and Defined Types
B_8_BIT_640x400	Global Variables, Constants, and Defined Types
B_8_BIT_640x480	Global Variables, Constants, and Defined Types
B_8_BIT_800x600	Global Variables, Constants, and Defined Types
B_8_BIT_DAC	BScreen

:

:	BPictureButton

<

Highlight()	BMenuItem

=

=	BFont
=	BPoint
=	BPolygon
=	BRect
=	BRegion
==	BFont
==	BPoint
==	BRect

_

_flags	BWindow
_frame	BWindow
_pulse	BWindow
_sizel	BWindow
_title	BWindow
_type	BWindow
_views	BWindow
_wfeel	BWindow
_wlook	BWindow
_wspace	BWindow
_zoom	BWindow

A

AcceptsDrop()	BTextView
AcceptsPaste()	BTextView
Activate()	BWindow
activate_workspace()	Global Functions
AddChild()	BBitmap
AddChild()	BWindow
AddChild()	View Hierarchy Functions
AddItem()	BListView
AddItem()	BMenu
AddItem()	BOutlineListView
AddLine()	BView Drawing-Related Functions
AddList()	BListView
AddList()	BOutlineListView
AddPoints()	BPolygon
AddReplicant()	BShelf
AddSeparatorItem()	BMenu
AddShape()	BShape
AddShortcut()	BWindow
AddTab()	BTabView
AddToSubset()	BWindow
AddUnder()	BOutlineListView
Adding a Scroll Bar to a BListView	BListView

AdjustReplicantBy()	BShelf
BAlert	BAlert
BAlert()	BAlert
AlertPosition()	BAlert
alert_type	BAlert
alert_type	Global Variables, Constants, and Defined Types
B_ALIGN_CEN	BMenuField
B_ALIGN_CEN	BStringView
B_ALIGN_CEN	BTextControl
B_ALIGN_CEN	BTextView
B_ALIGN_CENTER	Global Variables, Constants, and Defined Types
B_ALIGN_LEFT	BMenuField
B_ALIGN_LEFT	BStringView
B_ALIGN_LEFT	BTextControl
B_ALIGN_LEFT	BTextView
B_ALIGN_LEFT	Global Variables, Constants, and Defined Types
B_ALIGN_RIGH	BMenuField
B_ALIGN_RIGH	BStringView
B_ALIGN_RIGH	BTextControl
B_ALIGN_RIGH	BTextView
B_ALIGN_RIGHT	Global Variables, Constants, and Defined Types
Alignment()	BMenuField
Alignment()	BStringView

Alignment()	BTextView
alignment Constants	Global Variables, Constants, and Defined Types
alignment	Global Variables, Constants, and Defined Types
AllAttached()	BMenuField
AllAttached()	BView Hook Functions
AllDetached()	BView Hook Functions
B_ALL_WORKSPACES	BWindow
B_ALL_WORKSPACES	Global Variables, Constants, and Defined Types
AllowChar()	BTextView
AllowsDragging()	BShelf
AllowsZombies()	BShelf
AppendToPicture()	BView Drawing-Related Functions
Archive()	BBitmap
Archive()	BColorControl
Archive()	BControl
Archive()	BDragger
Archive()	BMenuBar
Archive()	BMenuField
Archive()	BMenu
Archive()	BOutlineListView
Archive()	BPopUpMenu
Archive()	BRadioButton
Archive()	BScrollBar

Archive()	BScrollView
Archive()	BShelf
Archive()	BStatusBar
Archive()	BStringItem
Archive()	BStringView
Archive()	BTabView
Archive()	BTab
Archive()	BTextControl
Archive()	BTextView
Archive()	BWindow
Archived Fields	BAlert
Archived Fields	BBitmap
Archived Fields	BBox
Archived Fields	BButton
Archived Fields	BCheckBox
Archived Fields	BColorControl
Archived Fields	BControl
Archived Fields	BDragger
Archived Fields	BListItem
Archived Fields	BListView
Archived Fields	BMenuBar
Archived Fields	BMenuField
Archived Fields	BMenuItem
Archived Fields	BMenu
Archived Fields	BPictureButton
Archived Fields	BPicture

Archived Fields	BScrollBar
Archived Fields	BScrollView
Archived Fields	BShape
Archived Fields	BSlider
Archived Fields	BStatusBar
Archived Fields	BStringItem
Archived Fields	BStringView
Archived Fields	BTabView
Archived Fields	BTab
Archived Fields	BTextControl
Archived Fields	BTextView
Archived Fields	BWindow
Archived Fields	General BView Functions
Archived Fields	Scripting and Archival
AreDraggersDrawn()	BDragger
AreTriggersEnabled()	BMenu
ascent	BFont
AsyncAutoDestruct()	BPopUpMenu
Asynchronous Alerts	BAlert
B_ASYNCHRONOUS_CONTROLS	BWindow
AttachedToWindow()	BBox
AttachedToWindow()	BButton
AttachedToWindow()	BColorControl
AttachedToWindow()	BControl
AttachedToWindow()	BDragger
AttachedToWindow()	BListView

AttachedToWindow()	BMenuBar
AttachedToWindow()	BMenuField
AttachedToWindow()	BMenu
AttachedToWindow()	BRadioButton
AttachedToWindow()	BScrollBar
AttachedToWindow()	BScrollView
AttachedToWindow()	BSlider
AttachedToWindow()	BStatusBar
AttachedToWindow()	BStringView
AttachedToWindow()	BTabView
AttachedToWindow()	BTextControl
AttachedToWindow()	BTextView
AttachedToWindow()	BView Hook Functions
Attaching to a Window	Drawing
Autoscrolling and Dragging Items	BListView
B_AVOID_FOCUS	BWindow
B_AVOID_FRONT	BWindow

B

B_BACKSPACE	Global Variables, Constants, and Defined Types
The Background Bitmap	Drawing
background_color	Global Functions
BarColor()	BSlider

BarColor()	BStatusBar
BarFrame()	BSlider
BarHeight()	BStatusBar
BarThickness()	BSlider
be_bold_font	BFont
be_fixed_font	BFont
be_plain_font	BFont
BeginJob()	BPrintJob
BeginLineArray()	BView Drawing-Related Functions
BeginPicture()	BView Drawing-Related Functions
BeginRectTracking()	Input-Related Functions
Behavior()	BPictureButton
B_BEVEL_JOIN	Graphics State Functions
BezierTo()	BShape
B_BIG_RGB_16_BIT	Global Variables, Constants, and Defined Types
B_BIG_RGB_32_BIT	Global Variables, Constants, and Defined Types
BBitmap	BBitmap
BBitmap()	BBitmap
~BBitmap()	BBitmap
Bitmap Data	BBitmap
B_BITMAP_SPACING	Global Variables, Constants, and Defined Types
Bitmaps and Patterns	Drawing

bitmaps_support_space()	Global Functions
Bits()	BBitmap
BitsLength()	BBitmap
B_BLANK_PEDESTAL	BScreen
Blending modes	Drawing
B_BLOCK_THUMB	BSlider
Blocks()	BFont
Border()	BBox
Border()	BMenuBar
Border()	BScrollView
B_BORDER_CONTENTS	BMenuBar
B_BORDER_CONTENTS	Global Variables, Constants, and Defined Types
B_BORDER_EACH_ITEM	BMenuBar
B_BORDER_EACH_ITEM	Global Variables, Constants, and Defined Types
B_BORDER_FRAME	BMenuBar
B_BORDER_FRAME	Global Variables, Constants, and Defined Types
border_style Constants	Global Variables, Constants, and Defined Types
B_BORDERED_WINDOW	BWindow
B_BORDERED_WINDOW	Global Variables, Constants, and Defined Types
B_BORDERED_WINDOW_LOOK	BWindow
BoundingBox()	BFont
Bounds()	BBitmap

Bounds()	BShape
Bounds()	BWindow
Bounds()	General BView Functions
BBox	BBox
BBox()	BBox
~BBox()	BBox
B_BUTT_CAP	Graphics State Functions
B_BUTT_JOIN	Graphics State Functions
ButtonAt()	BAlert
BButton	BButton
BButton()	BButton
~BButton()	BButton
button_spacing	BAlert
button_width Constants	Global Variables, Constants, and Defined Types
button_width	Global Variables, Constants, and Defined Types
ByteAt()	BTextView
BytesPerRow()	BBitmap

C

B_CMAP8	Global Variables, Constants, and Defined Types
B_CMYA32	Global Variables, Constants, and Defined Types
B_CMYK32	Global Variables,

	Constants, and Defined Types
B_CMY24	Global Variables, Constants, and Defined Types
B_CMY32	Global Variables, Constants, and Defined Types
CanAcceptReplicantMessage()	BShelf
CanAcceptReplicantView()	BShelf
CanContinue()	BPrintJob
CanEndLine()	BTextView
CancelJob()	BPrintJob
Cancellation	BPrintJob
B_CAPS_LOCK	Global Variables, Constants, and Defined Types
B_CAPS_SHIFT_TABLE	Global Variables, Constants, and Defined Types
B_CAPS_TABLE	Global Variables, Constants, and Defined Types
CellSize()	BColorControl
B_CHAR_SPACING	Global Variables, Constants, and Defined Types
Character Constants	Global Variables, Constants, and Defined Types
Character Encoding	Listening to the User
BCheckBox	BCheckBox
BCheckBox()	BCheckBox
~BCheckBox()	BCheckBox
ChildAt()	BBitmap

ChildAt()	BWindow
ChildAt()	View Hierarchy Functions
The CityItem Declaration	BListItem: Creating a Custom List Item
The CityItem Definition	BListItem: Creating a Custom List Item
Clear()	BShape
Clear()	BTextView
ClearViewBitmap()	BView Drawing-Related Functions
ClearViewOverlay()	BView Drawing-Related Functions
click_to_open	Global Functions
ClipToInversePicture()	Graphics State Functions
ClipToPicture()	Graphics State Functions
Clipping Functions	Graphics State Functions
The Clipping Region	Drawing
Close()	BShape
Collapse()	BOutlineListView
BColorControl	BColorControl
BColorControl()	BColorControl
~BColorControl()	BColorControl
ColorForIndex()	BScreen
Color	Graphics State Functions
ColorMap()	BScreen
ColorSpace()	BBitmap
ColorSpace()	BScreen

ColorSpace()	BTextView
Color Spaces	Drawing
B_COLOR_8_BIT	Global Variables, Constants, and Defined Types
color_control_layout	BColorControl
color_map	Global Variables, Constants, and Defined Types
color_space	Global Variables, Constants, and Defined Types
color_which constants	Global Variables, Constants, and Defined Types
Colors	Drawing
B_COMMAND_KEY	Global Variables, Constants, and Defined Types
CommitJob()	BPrintJob
CommitViewTransaction()	BWindow
A Complete Printing Function	BPrintJob
ConfigJob()	BPrintJob
ConfigPage()	BPrintJob
Constants and Defined Types	BAlert
Constants and Defined Types	BWindow
Constants	BFont
Constants	Global Variables, Constants, and Defined Types
ConstrainClippingRegion()	Graphics State Functions
ConstrainTo()	BPoint
Construction and Deletion	BAlert

Constructor and Destructor	BBitmap
Constructor and Destructor	BBox
Constructor and Destructor	BButton
Constructor and Destructor	BCheckBox
Constructor and Destructor	BColorControl
Constructor and Destructor	BControl
Constructor and Destructor	BDragger
Constructor and Destructor	BListItem
Constructor and Destructor	BListView
Constructor and Destructor	BMenuBar
Constructor and Destructor	BMenuField
Constructor and Destructor	BMenuItem
Constructor and Destructor	BMenu
Constructor and Destructor	BOutlineListView
Constructor and Destructor	BPictureButton
Constructor and Destructor	BPicture
Constructor and Destructor	BPolygon
Constructor and Destructor	BPopUpMenu
Constructor and Destructor	BPrintJob
Constructor and Destructor	BRadioButton
Constructor and Destructor	BRegion
Constructor and Destructor	BScreen
Constructor and Destructor	BScrollBar
Constructor and Destructor	BScrollView
Constructor and Destructor	BShapeIterator
Constructor and Destructor	BShape

Constructor and Destructor	BShelf
Constructor and Destructor	BSlider
Constructor and Destructor	BStatusBar
Constructor and Destructor	BStringItem
Constructor and Destructor	BStringView
Constructor and Destructor	BTabView
Constructor and Destructor	BTab
Constructor and Destructor	BTextControl
Constructor and Destructor	BTextView
Constructor and Destructor	BWindow
Constructor and Destructor	General BView Functions
Constructor	BAlert
Constructor	BFont
Constructor	BPoint
Constructor	BRect
Contains()	BRect
Contains()	BRegion
ContentLocation()	BMenuItem
BControl	BControl
BControl()	BControl
~BControl()	BControl
Control Values	Global Variables, Constants, and Defined Types
B_CONTROL_KEY	Global Variables, Constants, and Defined Types
B_CONTROL_OFF	Global Variables,

	Constants, and Defined Types
B_CONTROL_ON	Global Variables, Constants, and Defined Types
B_CONTROL_TABLE	Global Variables, Constants, and Defined Types
ConvertFromParent()	General BView Functions
ConvertFromScreen()	BWindow
ConvertFromScreen()	General BView Functions
ConvertToParent()	General BView Functions
ConvertToScreen()	BWindow
ConvertToScreen()	General BView Functions
Coordinate Data Types	The Coordinate Space
The Coordinate Space	The Coordinate Space
The Coordinate Space	The Coordinate Space
Coordinate Systems	The Coordinate Space
Coordination	BScrollBar
Copy()	BTextView
CopyBits()	BView Drawing-Related Functions
Copy mode	Drawing
CountChildren()	BBitmap
CountChildren()	BWindow
CountChildren()	View Hierarchy Functions
CountItems()	BListView
CountItems()	BMenu
CountItemsUnder()	BOutlineListView

CountLines()	BTextView
CountPoints()	BPolygon
CountRects()	BRegion
CountReplicants()	BShelf
CountTuned()	BFont
count_font_families()	Global Functions
count_font_styles()	Global Functions
count_workspaces()	Global Functions
Creating a BShape	BShape
Creating a BTabView	BTabView
Creating and Running an Alert Panel	BAlert
Creating and Using a BWindow	BWindow
CurrentFocus()	BWindow
CurrentLine()	BTextView
CurrentSelection()	BListView
CurrentValue()	BStatusBar
B_CURRENT_WORKSPACE	BWindow
current_workspace()	Global Functions
B_CURRENT_WORKSPACE	Global Variables, Constants, and Defined Types
Cursor Transit Constants	Global Variables, Constants, and Defined Types
Customizing a BSlider	BSlider
Customizing the Appearance of a BTab	BTab
Customizing through Virtual	BSlider: Customizing your

Functions	Slider
Cut()	BTextView

D

B_DELETE	Global Variables, Constants, and Defined Types
B_DPMS	BScreen
DPMSCapabilities()	BScreen
DPMSState()	BScreen
B_DARKEN_1_TINT	Global Variables, Constants, and Defined Types
B_DARKEN_2_TINT	Global Variables, Constants, and Defined Types
B_DARKEN_3_TINT	Global Variables, Constants, and Defined Types
B_DARKEN_4_TINT	Global Variables, Constants, and Defined Types
B_DARKEN_MAX_TINT	Global Variables, Constants, and Defined Types
Data Members	BPoint
Data Members	BRect

Dead-Key Mapping	Global Variables, Constants, and Defined Types
DefaultButton()	BWindow
Defined Types	BFont
Defined Types	Global Variables, Constants, and Defined Types
Delete()	BTextView
DeleteReplicant()	BShelf
DeleteText()	BTextView
Derived Classes	BControl
descent	BFont
DeselectAll()	BListView
Deselect()	BListItem
Deselect()	BListView
Deselect()	BTab
DeselectExcept()	BListView
DesktopColor()	BScreen
B_DESKTOP_COLOR	Global Variables, Constants, and Defined Types
DetachedFromWindow()	BDragger
DetachedFromWindow()	BTextView
DetachedFromWindow()	BView Hook Functions
Developing a BView Subclass	BView Introduction
Direction()	BFont

DisableUpdates()	BWindow
B_DISABLE_ANTIALIASING	Global Variables, Constants, and Defined Types
Disabled Items	BMenuItem
DisabledOff()	BPictureButton
DisabledOn	BPictureButton
B_DISABLED_LABEL_TINT	Global Variables, Constants, and Defined Types
B_DISABLED_MARK_TINT	Global Variables, Constants, and Defined Types
DisallowChar()	BTextView
DispatchMessage()	BWindow
Divider()	BMenuField
Divider()	BTextControl
DoForEach()	BListView
B_DOCUMENT_WINDOW	BWindow
B_DOCUMENT_WINDOW	Global Variables, Constants, and Defined Types
B_DOCUMENT_WINDOW_LOOK	BWindow
DoesAutoindent()	BTextView
DoesUndo()	BTextView
DoesWordWrap()	BTextView
double_arrows	Global Functions

B_DOWN_ARROW	Global Variables, Constants, and Defined Types
B_DPMS_OFF	BScreen
B_DPMS_ON	BScreen
B_DPMS_STAND_BY	BScreen
B_DPMS_SUSPEND	BScreen
Drag and Drop	Listening to the User
DragMessage()	Input-Related Functions
BDragger	BDragger
BDragger()	BDragger
~BDragger()	BDragger
DrawAfterChildren()	BView Hook Functions
Draw()	BBox
Draw()	BButton
Draw()	BColorControl
Draw()	BDragger
Draw()	BListView
Draw()	BMenuBar
Draw()	BMenuField
Draw()	BMenuItem
Draw()	BMenu
Draw()	BRadioButton
Draw()	BScrollView
Draw()	BSlider
Draw()	BStatusBar

Draw()	BStringView
Draw()	BTabView
Draw()	BTextControl
Draw()	BTextView
Draw()	BView Hook Functions
DrawBar()	BSlider
DrawBitmapAsync()	BView Drawing-Related Functions
DrawBitmap()	BView Drawing-Related Functions
DrawBox()	BTabView
DrawChar()	BView Drawing-Related Functions
DrawContent()	BMenuItem
DrawFocusMark()	BSlider
DrawFocusMark()	BTab
DrawHashMarks()	BSlider
DrawItem()	BListItem
DrawItem()	BStringItem
DrawLabel()	BTab
DrawPictureAsync()	BView Drawing-Related Functions
DrawPicture()	BView Drawing-Related Functions
DrawSlider()	BSlider
DrawString()	BView Drawing-Related Functions

DrawTab()	BTab
DrawTabs()	BTabView
DrawText()	BSlider
DrawThumb()	BSlider
DrawView()	BPrintJob
Drawing and Message Handling in the View Hierarchy	The Interface Kit
Drawing Coordinates	BPrintJob
Drawing	Drawing
Drawing	Drawing
Drawing during an Update	Drawing
Drawing in a View	The Interface Kit
DrawingMode()	Graphics State Functions
Drawing Modes and Color Spaces	Drawing
Drawing Modes	Drawing
Drawing Modes	Graphics State Functions
Drawing on the Page	BPrintJob
Drawing outside of an Update	Drawing
Drawing the Focus Indicator	Listening to the User
drawing_mode Constants	Global Variables, Constants, and Defined Types
drawing_mode	Global Variables, Constants, and Defined Types

Dropping into the View Hierarchy	BShelf

E

B_END	Global Variables, Constants, and Defined Types
B_ENTER	Global Variables, Constants, and Defined Types
B_ESCAPE	Global Variables, Constants, and Defined Types
EachItemUnder()	BOutlineListView
edge_info	BFont
edge_info	Global Variables, Constants, and Defined Types
Edges	BFont
EnableUpdates()	BWindow
EnabledOff()	BPictureButton
EnabledOn()	BPictureButton
The Enabled Property	BControl
The Enabled Property	BMenu
Encoding()	BFont
EndLineArray()	BView Drawing-Related Functions
EndPicture()	BView Drawing-Related Functions
EndRectTracking()	Input-Related Functions
B_ENTERED_VIEW	BView Hook Functions
B_ENTERED_VIEW	Global Variables, Constants, and Defined Types

Erasing the Clipping Region	Drawing
escapement_delta	BFont
escapement_delta	Global Variables, Constants, and Defined Types
Escapements	BFont
EventMask()	Input-Related Functions
Exclude()	BRegion
B_EXITED_VIEW	BView Hook Functions
B_EXITED_VIEW	Global Variables, Constants, and Defined Types
Expand()	BOutlineListView

F

B_F10_KEY	Global Variables, Constants, and Defined Types
B_F11_KEY	Global Variables, Constants, and Defined Types
B_F12_KEY	Global Variables, Constants, and Defined Types
B_F1_KEY	Global Variables, Constants, and Defined Types
B_F2_KEY	Global Variables, Constants, and Defined

	Types
B_F3_KEY	Global Variables, Constants, and Defined Types
B_F4_KEY	Global Variables, Constants, and Defined Types
B_F5_KEY	Global Variables, Constants, and Defined Types
B_F6_KEY	Global Variables, Constants, and Defined Types
B_F7_KEY	Global Variables, Constants, and Defined Types
B_F8_KEY	Global Variables, Constants, and Defined Types
B_F9_KEY	Global Variables, Constants, and Defined Types
Face()	BFont
FamilyAndStyle()	BFont
B_FANCY_BORDER	Global Variables, Constants, and Defined Types
"Feel"	BWindow
FileFormat()	BFont

FillArc()	BView Drawing-Related Functions
FillBezier()	BView Drawing-Related Functions
FillColor()	BSlider
FillEllipse()	BView Drawing-Related Functions
FillPolygon()	BView Drawing-Related Functions
FillRect()	BView Drawing-Related Functions
FillRegion()	BView Drawing-Related Functions
FillRoundRect()	BView Drawing-Related Functions
FillShape()	BView Drawing-Related Functions
FillTriangle()	BView Drawing-Related Functions
Filling and Stroking Polygons	The Coordinate Space
Filling and Stroking Rectangles	The Coordinate Space
FindItem()	BMenu
FindMarked()	BMenu
FindView()	BBitmap
FindView()	BWindow
FindView()	View Hierarchy Functions
FindWord()	BTextView
FirstItem()	BListView

FirstPage()	BPrintJob
B_FIXED_SPACING	Global Variables, Constants, and Defined Types
Flags()	BFont
"Flags"	BWindow
Flags()	General BView Functions
FlattenRunArray()	BTextView
B_FLOATING_ALL_WINDOW_FEEL	BWindow
B_FLOATING_APP_WINDOW_FEEL	BWindow
B_FLOATING_SUBSET_WINDOW_FEEL	BWindow
B_FLOATING_WINDOW	BWindow
B_FLOATING_WINDOW_LOOK	BWindow
Flush()	BView Drawing-Related Functions
Flush()	BWindow
Focus	BView Introduction
FocusTab()	BTabView
Focus View Commands	Listening to the User
B_FOLLOW_ALL	Global Variables, Constants, and Defined Types
B_FOLLOW_ALL_SI	General BView Functions
B_FOLLOW_BOTTOM	General BView Functions
B_FOLLOW_BOTTOM	Global Variables, Constants, and Defined Types

B_FOLLOW_H_CENT	General BView Functions
B_FOLLOW_H_CENTER	Global Variables, Constants, and Defined Types
B_FOLLOW_LEFT	General BView Functions
B_FOLLOW_LEFT	Global Variables, Constants, and Defined Types
B_FOLLOW_LEFT_RI	General BView Functions
B_FOLLOW_LEFT_RIGHT	Global Variables, Constants, and Defined Types
B_FOLLOW_NONE	General BView Functions
B_FOLLOW_NONE	Global Variables, Constants, and Defined Types
B_FOLLOW_RIGHT	General BView Functions
B_FOLLOW_RIGHT	Global Variables, Constants, and Defined Types
B_FOLLOW_TOP	General BView Functions
B_FOLLOW_TOP	Global Variables, Constants, and Defined Types
B_FOLLOW_TOP_BOTT	General BView Functions
B_FOLLOW_TOP_BOTTOM	Global Variables, Constants, and Defined Types

B_FOLLOW_V_CENTER	General BView Functions
B_FOLLOW_V_CENTER	Global Variables, Constants, and Defined Types
BFont	BFont
BFont()	BFont
Font Encodings	Global Variables, Constants, and Defined Types
Font Flags	Global Variables, Constants, and Defined Types
font	Global Functions
Font Name Lengths	Global Variables, Constants, and Defined Types
Font Properties	Global Variables, Constants, and Defined Types
Font Spacing Modes	Global Variables, Constants, and Defined Types
B_FONT_ALL	Global Variables, Constants, and Defined Types
font_direction	BFont
font_direction Constants	Global Variables, Constants, and Defined Types

font_direction	Global Variables, Constants, and Defined Types
B_FONT_ENCODING	Global Variables, Constants, and Defined Types
B_FONT_FACE	Global Variables, Constants, and Defined Types
font_family	BFont
font_family	Global Variables, Constants, and Defined Types
B_FONT_FAMILY_AND_STYLE	Global Variables, Constants, and Defined Types
B_FONT_FAMILY_NAME_LENGTH	Global Variables, Constants, and Defined Types
font_file_format	BFont
B_FONT_FLAGS	Global Variables, Constants, and Defined Types
font_height	BFont
font_height	Global Variables, Constants, and Defined Types
B_FONT_LEFT_TO_RIGHT	Global Variables, Constants, and Defined

	Types
font_metric_mode	BFont
B_FONT_RIGHT_TO_LEFT	Global Variables, Constants, and Defined Types
B_FONT_ROTATION	Global Variables, Constants, and Defined Types
B_FONT_SHEAR	Global Variables, Constants, and Defined Types
font_size	Global Functions
B_FONT_SIZE	Global Variables, Constants, and Defined Types
B_FONT_SPACING	Global Variables, Constants, and Defined Types
font_style	BFont
font_style	Global Variables, Constants, and Defined Types
B_FONT_STYLE_NAME_LENGTH	Global Variables, Constants, and Defined Types
ForceFontAliasing()	Graphics State Functions
Forcing an Update	Drawing
Forcing an Update while Responding to an Event	Drawing

Frame and Bounds Rectangles	Drawing
Frame()	BMenuItem
Frame()	BPolygon
Frame()	BRegion
Frame()	BScreen
"Frame"	BWindow
Frame()	General BView Functions
FrameMoved()	BView Hook Functions
FrameMoved()	BWindow
FrameResized()	BBox
FrameResized()	BListView
FrameResized()	BSlider
FrameResized()	BTextView
FrameResized()	BView Hook Functions
FrameResized()	BWindow
"Frame"	Scripting and Archival
B_FRAME_EVENTS	General BView Functions
B_FRAME_EVENTS	Global Variables, Constants, and Defined Types
Framework for the User Interface	The Interface Kit
FullListCountItems()	BOutlineListView
FullListCurrentSelection()	BOutlineListView
FullListDoForEach()	BOutlineListView
FullListFirstItem()	BOutlineListView
FullListHasItem()	BOutlineListView

FullListIndexOf()	BOutlineListView
FullListIsEmpty()	BOutlineListView
FullListItemAt()	BOutlineListView
FullListLastItem()	BOutlineListView
FullListSortItems()	BOutlineListView
B_FULL_UPDATE_ON_RESIZE	General BView Functions
B_FULL_UPDATE_ON_RESIZE	Global Variables, Constants, and Defined Types
Function Key Constants	Global Variables, Constants, and Defined Types
B_FUNCTION_KEY	Global Variables, Constants, and Defined Types
Functions	Global Functions

G

B_GRAY1	Global Variables, Constants, and Defined Types
B_GRAY8	Global Variables, Constants, and Defined Types
General and Miscellaneous	Graphics State Functions
General BView Functions	General BView Functions
General BView Functions	General BView Functions
GetAlignment()	BTextControl
GetBitmap()	BScreen

GetBlendingMode()	Graphics State Functions
GetBoundingBoxesAsGlyphs()	BFont
GetBoundingBoxesAsString()	BFont
GetBoundingBoxesForStrings()	BFont
GetClippingRegion()	Graphics State Functions
GetContentSize()	BMenuItem
GetDeviceInfo	BScreen
GetDragParameters()	BTextView
GetEdges()	BFont
GetEscapements()	BFont
GetFamilyAndStyle()	BFont
GetFontAndColor()	BTextView
GetFont()	Graphics State Functions
GetFontHeight()	Graphics State Functions
GetGlyphShapes()	BFont
GetHasGlyphs()	BFont
GetHeight()	BFont
GetItemMargins()	BMenu
GetMode()	BScreen
GetModeList()	BScreen
GetMouse()	Input-Related Functions
GetPixelClockLimits()	BScreen
GetPreferredSize()	BButton
GetPreferredSize()	BColorControl
GetPreferredSize()	BRadioButton
GetPreferredSize()	BSlider

GetPreferredSize()	BTextControl
GetPreferredSize()	BView Hook Functions
GetRange()	BScrollBar
GetResolution()	BPrintJob
GetSelection()	BTextView
GetSizeLimits()	BWindow
GetSteps()	BScrollBar
GetStringWidths()	BFont
GetStringWidths()	Graphics State Functions
GetSupportedSuites()	BControl
GetSupportedSuites()	BWindow
GetText()	BTextView
GetTextRegion()	BTextView
GetTimingConstraints()	BScreen
GetTruncatedStrings()	BFont
GetTunedInfo()	BFont
GetWindowAlignment()	BWindow
get_deskbar_frame()	Global Functions
get_font_family()	Global Functions
get_font_style()	Global Functions
get_menu_info()	Global Functions
get_pixel_size_for()	Global Functions
get_scroll_bar_info()	Global Functions
Global Functions	Global Functions
Global Functions	Global Functions
Global Objects	BPoint

Global Variables	Global Variables, Constants, and Defined Types
Global Variables, Constants, and Defined Types	Global Variables, Constants, and Defined Types
Global Variables, Constants, and Defined Types	Global Variables, Constants, and Defined Types
Go()	BAlert
Go()	BPopUpMenu
GoToLine()	BTextView
Graphics Environment	Drawing
Graphics Primitives	BTextView
Graphics State Functions	Graphics State Functions
Graphics State Functions	Graphics State Functions
B_GRAYSCALE_8_BIT	Global Variables, Constants, and Defined Types

---

H

B_HLSA32	Global Variables, Constants, and Defined Types
B_HLS24	Global Variables, Constants, and Defined Types
B_HLS32	Global Variables, Constants, and Defined Types
B_HOME	Global Variables,

	Constants, and Defined Types
B_HORIZONTAL	Global Variables, Constants, and Defined Types
B_HSIA32	Global Variables, Constants, and Defined Types
B_HSI24	Global Variables, Constants, and Defined Types
B_HSI32	Global Variables, Constants, and Defined Types
B_HSVA32	Global Variables, Constants, and Defined Types
B_HSV24	Global Variables, Constants, and Defined Types
B_HSV32	Global Variables, Constants, and Defined Types
B_H_SCROLL_BAR_HEIGHT	Global Variables, Constants, and Defined Types
Handling Keyboard Actions	Listening to the User
Handling Messages in a View	The Interface Kit
B_HARDWARE_CURSOR	BScreen

HasItem()	BListView
B_HAS_TUNED_FONT	Global Variables, Constants, and Defined Types
HashMarkCount()	BSlider
HashMarks()	BSlider
HashMarksFrame()	BSlider
B_HASH_MARKS_BOTH	BSlider
B_HASH_MARKS_BOTTOM	BSlider
B_HASH_MARKS_LEFT	BSlider
B_HASH_MARKS_NONE	BSlider
B_HASH_MARKS_RIGHT	BSlider
B_HASH_MARKS_TOP	BSlider
Height()	BListItem
Height()	BRect
"Hidden"	BWindow
"Hidden"	Scripting and Archival
HideAllDraggers()	BDragger
Hide()	BMenuBar
Hide()	BMenu
Hide()	BWindow
Hide()	General BView Functions
HideTyping()	BTextView
HighColor()	Graphics State Functions
Highlight()	BMenuItem
Highlight()	BTextView

B_HIGHLIGHT_BACKGROUND_TINT	Global Variables, Constants, and Defined Types
Hook Functions	BButton
Hook Functions	BControl
Hook Functions	BListItem
Hook Functions	BListView
Hook Functions	BMenuItem
Hook Functions	BMenu
Hook Functions	BScrollBar
Hook Functions	BShelf
Hook Functions	BSlider
Hook Functions	BTabView
Hook Functions	BTab
Hook Functions	BTextView
Hook Functions	BView Hook Functions
Hook Functions	BWindow

I

ID()	BScreen
The ID Property	BShelf
B_INSERT	Global Variables, Constants, and Defined Types
idle_time()	Global Functions
Include()	BRegion
IndexForColor()	BScreen

IndexOf()	BListView
IndexOf()	BMenu
IndexOf()	BShelf
Inherited Functions	BOutlineListView
InitiateDrag()	BListView
Input-Related Functions	Input-Related Functions
Input-Related Functions	Input-Related Functions
Insert()	BTextView
InsertText()	BTextView
InsetBy()	BRect
InsetByCopy()	BRect
InsetBySelf()	BRect
B_INSIDE_VIEW	BView Hook Functions
B_INSIDE_VIEW	Global Variables, Constants, and Defined Types
Instantiate()	BBitmap
Instantiate()	BColorControl
Instantiate()	BControl
Instantiate()	BDragger
Instantiate()	BListView
Instantiate()	BMenuBar
Instantiate()	BMenuField
Instantiate()	BMenu
Instantiate()	BOutlineListView
Instantiate()	BPopUpMenu
Instantiate()	BRadioButton

Instantiate()	BScrollBar
Instantiate()	BScrollView
Instantiate()	BShape
Instantiate()	BShelf
Instantiate()	BStatusBar
Instantiate()	BStringItem
Instantiate()	BTabView
Instantiate()	BTab
Instantiate()	BTextControl
Instantiate()	BTextView
Instantiate()	General BView Functions
IntegerHeight()	BRect
IntegerWidth()	BRect
The Interface Kit	The Interface Kit
The Interface Kit	The Interface Kit
Interface Messages	Global Variables, Constants, and Defined Types
Interface Messages	Listening to the User
IntersectWith()	BRegion
Intersects()	BRect
Intersects()	BRegion
Invalidate()	BView Drawing-Related Functions
InvalidateItem()	BListView
InvalidateLayout()	BMenu
InvertRect()	BView Drawing-Related Functions
InvocationCommand()	BListView

InvocationMessage()	BListView
Invoke()	BControl
Invoke()	BListView
Invoke()	BMenuItem
B_IO_FB_NA	BScreen
IsActive()	BWindow
IsBorderHighlighted()	BScrollView
IsDefault()	BButton
IsDirty()	BShelf
IsEditable()	BTextView
IsEmpty()	BListView
IsEnabled()	BControl
IsEnabled()	BListItem
IsEnabled()	BMenuField
IsEnabled()	BMenuItem
IsEnabled()	BMenu
IsEnabled()	BTab
IsExpanded()	BListItem
IsExpanded()	BOutlineListView
IsFixed()	BFont
IsFloating()	BWindow
IsFocus()	BTab
IsFocusChanging()	BControl
IsFocus()	General BView Functions
IsFront()	BWindow
IsFullAndHalfFixed()	BFont

IsHidden()	BWindow
IsHidden()	General BView Functions
IsItemSelected()	BListView
IsLabelFromMarked()	BMenu
IsLocked()	BBitmap
IsMarked()	BMenuItem
IsMinimized()	BWindow
IsModal()	BWindow
IsPrinting()	General BView Functions
IsRadioMode()	BMenu
IsResizable()	BTextView
IsSelectable()	BTextView
IsSelected()	BListItem
IsSelected()	BMenuItem
IsSelected()	BTab
IsSettingsMessageValid()	BPrintJob
IsStylable()	BTextView
IsTypeEnforced()	BShelf
IsTypingHidden()	BTextView
IsValid()	BBitmap
IsValid()	BRect
IsValid()	BScreen
IsVisibilityChanging()	BDragger
B_IS_FIXED	Global Variables, Constants, and Defined Types
B_ISO_8859_1	Global Variables,

	Constants, and Defined Types
B_ISO_8859_10	Global Variables, Constants, and Defined Types
B_ISO_8859_2	Global Variables, Constants, and Defined Types
B_ISO_8859_3	Global Variables, Constants, and Defined Types
B_ISO_8859_4	Global Variables, Constants, and Defined Types
B_ISO_8859_5	Global Variables, Constants, and Defined Types
B_ISO_8859_6	Global Variables, Constants, and Defined Types
B_ISO_8859_7	Global Variables, Constants, and Defined Types
B_ISO_8859_8	Global Variables, Constants, and Defined Types
B_ISO_8859_9	Global Variables, Constants, and Defined Types
ItemAt()	BListView
ItemAt()	BMenu
ItemFrame()	BListView
The Item Property	BListView
ItemUnderAt()	BOutlineListView
Items()	BListView
B_ITEMS_IN_COLUMN	BMenu
B_ITEMS_IN_COLUMN	BMenuBar
B_ITEMS_IN_COLUMN	Global Variables,

	Constants, and Defined Types
B_ITEMS_IN_MATRIX	BMenu
B_ITEMS_IN_MATRIX	BMenuBar
B_ITEMS_IN_MATRIX	Global Variables, Constants, and Defined Types
B_ITEMS_IN_ROW	BMenu
B_ITEMS_IN_ROW	Global Variables, Constants, and Defined Types
Iterate()	BShapeIterator
IterateBezierTo()	BShapeIterator
IterateClose()	BShapeIterator
IterateLineTo()	BShapeIterator
IterateMoveTo()	BShapeIterator

K

KeyDown()	BButton
KeyDown()	BColorControl
KeyDown()	BControl
KeyDown()	BListView
KeyDown()	BMenuField
KeyDown()	BMenu
KeyDown()	BOutlineListView
KeyDown()	BRadioButton
KeyDown()	BSlider
KeyDown()	BTabView

KeyDown()	BTextView
KeyDown()	BView Hook Functions
KeyIncrementValue()	BSlider
The Key Menu Bar	BMenuBar
KeyMenuBar()	BWindow
KeyUp()	BView Hook Functions
B_KEY_DOWN	Global Variables, Constants, and Defined Types
key_info	Global Variables, Constants, and Defined Types
key_map	Global Variables, Constants, and Defined Types
B_KEY_UP	Global Variables, Constants, and Defined Types
Keyboard Navigation	Listening to the User
keyboard_navigation_color()	Global Functions
B_KEYBOARD_NAVIGATION_COLOR	Global Variables, Constants, and Defined Types
A Kind of BMenu	BMenuBar
Kinds of Items	BMenuItem
Kinds of Keyboard Messages	Listening to the User
knob	Global Functions

L

B_LABA32	Global Variables, Constants, and Defined Types
B_LAB24	Global Variables, Constants, and Defined Types
B_LAB32	Global Variables, Constants, and Defined Types
Label()	BBox
Label()	BControl
Label()	BMenuField
Label()	BMenuItem
Label()	BStatusBar
Label()	BTab
The Label Property	BControl
The Label Property	BMenu
LabelView()	BBox
LastItem()	BListView
LastMouseMovedView()	BWindow
LastPage()	BPrintJob
Layout()	BColorControl
Layout()	BMenu
leading	BFont
LeftTop()	General BView Functions
B_LEFT_ARROW	Global Variables, Constants, and Defined Types
B_LEFT_COMMAND_KEY	Global Variables,

	Constants, and Defined Types
B_LEFT_CONTROL_KEY	Global Variables, Constants, and Defined Types
B_LEFT_OPTION_KEY	Global Variables, Constants, and Defined Types
B_LEFT_SHIFT_KEY	Global Variables, Constants, and Defined Types
B_LIGHTEN_1_TINT	Global Variables, Constants, and Defined Types
B_LIGHTEN_2_TINT	Global Variables, Constants, and Defined Types
B_LIGHTEN_MAX_TINT	Global Variables, Constants, and Defined Types
LineAt()	BTextView
LineCapMode()	Graphics State Functions
LineHeight()	BTextView
LineJoinMode()	Graphics State Functions
LineMiterLimit()	Graphics State Functions
LineTo()	BShape
LineWidth()	BTextView
BListItem	BListItem
BListItem()	BListItem
~BListItem()	BListItem
BListItem: Creating a Custom List Item	BListItem: Creating a Custom List Item
BListItem: Creating a Custom List Item	BListItem: Creating a Custom List Item

ListType()	BListView
BListView	BListView
BListView()	BListView
~BListView()	BListView
list_view_type Constants	Global Variables, Constants, and Defined Types
Listening to the User	Listening to the User
Listening to the User	Listening to the User
Lists and List Views	BListView
Lock()	BBitmap
Locking the Window	BView Introduction
Look and Feel	BAlert
"Look"	BWindow
LowColor()	Graphics State Functions

M

B_MINIMIZE	Global Variables, Constants, and Defined Types
B_MACINTOSH_ROMAN	Global Variables, Constants, and Defined Types
Main Screen	Global Variables, Constants, and Defined Types
MakeDefault()	BButton

MakeEditable()	BTextView
MakeEmpty()	BListView
MakeEmpty()	BOutlineListView
MakeEmpty()	BRegion
MakeFocus()	BControl
MakeFocus()	BListView
MakeFocus()	BMenuField
MakeFocus()	BTabView
MakeFocus()	BTab
MakeFocus()	BTextControl
MakeFocus()	BTextView
MakeFocus()	Input-Related Functions
MakeResizable()	BTextView
MakeSelectable()	BTextView
MapTo()	BPolygon
Mapping Coordinates to Pixels	The Coordinate Space
The Mark Property	BMenu
Marked Items	BMenuItem
MaxBytes()	BTextView
MaxContentWidth()	BMenu
MaxLimitLabel()	BSlider
MaxValue()	BStatusBar
The Mechanics of Drawing	Drawing
Member Functions	BAlert

Member Functions	BBitmap
Member Functions	BBox
Member Functions	BButton
Member Functions	BCheckBox
Member Functions	BColorControl
Member Functions	BControl
Member Functions	BDragger
Member Functions	BFont
Member Functions	BListItem
Member Functions	BListView
Member Functions	BMenuBar
Member Functions	BMenuField
Member Functions	BMenuItem
Member Functions	BMenu
Member Functions	BOutlineListView
Member Functions	BPictureButton
Member Functions	BPicture
Member Functions	BPoint
Member Functions	BPolygon
Member Functions	BPopUpMenu
Member Functions	BPrintJob
Member Functions	BRadioButton
Member Functions	BRect
Member Functions	BRegion
Member Functions	BScreen
Member Functions	BScrollBar

Member Functions	BScrollView
Member Functions	BShapeIterator
Member Functions	BShape
Member Functions	BShelf
Member Functions	BSlider
Member Functions	BStatusBar
Member Functions	BStringItem
Member Functions	BStringView
Member Functions	BTabView
Member Functions	BTab
Member Functions	BTextControl
Member Functions	BTextView
Member Functions	BWindow
Member Functions	General BView Functions
Menu()	BMenuField
Menu()	BMenuItem
BMenu	BMenu
BMenu()	BMenu
~BMenu()	BMenu
BMenuBar	BMenuBar
BMenuBar()	BMenuBar
~BMenuBar()	BMenuBar
MenuBar()	BMenuField
"MenuBar"	BWindow
BMenuField	BMenuField

BMenuField()	BMenuField
~BMenuField()	BMenuField
Menu Hierarchy	BMenu
MenuItem()	BMenuField
BMenuItem	BMenuItem
BMenuItem()	BMenuItem
~BMenuItem()	BMenuItem
The MenuItem Property	BMenu
Menu Items	BMenu
The Menu Property	BMenu
B_MENU_BACKGROUND_COLOR	Global Variables, Constants, and Defined Types
menu_bar_border Constants	Global Variables, Constants, and Defined Types
menu_bar_border	Global Variables, Constants, and Defined Types
menu_info	Global Functions
menu_info	Global Variables, Constants, and Defined Types
B_MENU_ITEM_TEXT_COLOR	Global Variables, Constants, and Defined Types
B_MENU_KEY	Global Variables, Constants, and

	Defined Types
menu_layout Constants	Global Variables, Constants, and Defined Types
menu_layout	Global Variables, Constants, and Defined Types
B_MENU_SELECTED_ITEM_TEXT_COLOR	Global Variables, Constants, and Defined Types
B_MENU_SELECTION_BACKGROUND_COLOR	Global Variables, Constants, and Defined Types
MenusBeginning()	BWindow
MenusEnded()	BWindow
Message Protocols	Listening to the User
MessageReceived()	BColorControl
MessageReceived()	BControl
MessageReceived()	BDragger
MessageReceived()	BStatusBar
MessageReceived()	BTextControl
MessageReceived()	BTextView
MessageReceived()	BWindow
MessageReceived()	Input-Related Functions
MinLimitLabel()	BSlider
min_knob_size	Global Functions

"Minimize"	BWindow
B_MITER_JOIN	Graphics State Functions
B_MODAL_ALL_WINDOW_FEEL	BWindow
B_MODAL_APP_WINDOW_FEEL	BWindow
B_MODAL_SUBSET_WINDOW_FEEL	BWindow
B_MODAL_WINDOW	BWindow
B_MODAL_WINDOW	Global Variables, Constants, and Defined Types
B_MODAL_WINDOW_LOOK	BWindow
Mode Definitions	Drawing
ModificationMessage()	BSlider
ModificationMessage()	BTextControl
Modifier States	Global Variables, Constants, and Defined Types
B_MONOCHROME_1_BIT	Global Variables, Constants, and Defined Types
Mouse Buttons	Global Variables, Constants, and Defined Types
MouseDown()	BButton
MouseDown()	BColorControl
MouseDown()	BDragger
MouseDown()	BListView
MouseDown()	BMenuBar

MouseDown()	BMenuField
MouseDown()	BOutlineListView
MouseDown()	BRadioButton
MouseDown()	BSlider
MouseDown()	BTabView
MouseDown()	BTextControl
MouseDown()	BTextView
MouseDown()	BView Hook Functions
MouseMoved()	BTextView
MouseMoved()	BView Hook Functions
MouseUp()	BView Hook Functions
B_MOUSE_DOWN	Global Variables, Constants, and Defined Types
mouse_map	Global Variables, Constants, and Defined Types
B_MOUSE_MOVED	Global Variables, Constants, and Defined Types
B_MOUSE_UP	Global Variables, Constants, and Defined Types
MoveBy()	BWindow
MoveBy()	View Hierarchy

	Functions
MoveItem()	BListView
MovePenBy()	Graphics State Functions
MovePenTo()	Graphics State Functions
MoveTo()	BShape
MoveTo()	BWindow
MoveTo()	View Hierarchy Functions
Multiple Screens	BScreen
B_MULTIPLE_SELECTION_LIST	Global Variables, Constants, and Defined Types

N

B_NAVIGABLE	General BView Functions
B_NAVIGABLE	Global Variables, Constants, and Defined Types
The Name Property	BShelf
B_NAVIGABLE_JUMP	General BView Functions
B_NAVIGABLE_JUMP	Global Variables, Constants, and Defined Types
NeedsUpdate()	BWindow
NextSibling()	View Hierarchy Functions

B_NO_BORDER	Global Variables, Constants, and Defined Types
B_NO_BORDER_WINDOW_LOOK	BWindow
B_NO_COLOR_SPACE	Global Variables, Constants, and Defined Types
B_NO_TINT	Global Variables, Constants, and Defined Types
B_NO_WORKSPACE_ACTIVATION	BWindow
Nonfractional Coordinates	Drawing
B_NORMAL_TABLE	Global Variables, Constants, and Defined Types
B_NORMAL_WINDOW_FEEL	BWindow
B_NOT_ANCHORED_ON_ACTIVATE	BWindow
B_NOT_CLOSABLE	BWindow
B_NOT_CLOSABLE	Global Variables, Constants, and Defined Types
B_NOT_H_RESIZABLE	BWindow
B_NOT_H_RESIZABLE	Global Variables, Constants, and Defined Types
B_NOT_MINIMIZABLE	BWindow
B_NOT_MINIMIZABLE	Global Variables, Constants, and Defined

	Types
B_NOT_MOVABLE	BWindow
B_NOT_MOVABLE	Global Variables, Constants, and Defined Types
B_NOT_RESIZABLE	BWindow
B_NOT_RESIZABLE	Global Variables, Constants, and Defined Types
B_NOT_V_RESIZABLE	BWindow
B_NOT_V_RESIZABLE	Global Variables, Constants, and Defined Types
B_NOT_ZOOMABLE	BWindow
B_NOT_ZOOMABLE	Global Variables, Constants, and Defined Types
B_NUM_LOCK	Global Variables, Constants, and Defined Types

O

B_ORIGIN	BPoint
OffscreenView()	BSlider
OffsetAt()	BTextView
OffsetBy()	BRect
OffsetBy()	BRegion

OffsetByCopy()	BRect
OffsetBySelf()	BRect
OffsetTo()	BRect
OffsetToCopy()	BRect
OffsetToSelf()	BRect
Offsets	BTextView
B_OP_ADD	Global Variables, Constants, and Defined Types
B_OP_ALPHA	Global Variables, Constants, and Defined Types
B_OP_BLEND	Global Variables, Constants, and Defined Types
B_OP_COPY	Global Variables, Constants, and Defined Types
B_OP_ERASE	Global Variables, Constants, and Defined Types
B_OP_INVERT	Global Variables, Constants, and Defined Types
B_OP_MAX	Global Variables, Constants, and Defined Types
B_OP_MIN	Global Variables, Constants, and Defined Types
B_OP_OVER	Global Variables, Constants, and Defined Types
B_OP_SELECT	Global Variables, Constants, and Defined Types
B_OP_SUBTRACT	Global Variables, Constants, and Defined Types

OpenViewTransaction()	BWindow
Operators	BFont
Operators	BPoint
Operators	BPolygon
Operators	BRect
Operators	BRegion
B_OPTION_CAPS_SHIFT_TABLE	Global Variables, Constants, and Defined Types
B_OPTION_CAPS_TABLE	Global Variables, Constants, and Defined Types
B_OPTION_KEY	Global Variables, Constants, and Defined Types
B_OPTION_SHIFT_TABLE	Global Variables, Constants, and Defined Types
B_OPTION_TABLE	Global Variables, Constants, and Defined Types
Orientation()	BScrollBar
Orientation()	BSlider
orientation Constants	Global Variables, Constants, and Defined Types
orientation	Global Variables, Constants, and Defined Types
Origin()	General BView Functions
Other BShelf Features	BShelf
Other Drawing Functions	BView Drawing-Related Functions
Other Keyboard Commands	Listening to the User

OutlineLevel()	BListItem
BOutlineListView	BOutlineListView
BOutlineListView()	BOutlineListView
~BOutlineListView()	BOutlineListView
Outline Structure	BOutlineListView
B_OUTLINE_RESIZE	BWindow
B_OUTSIDE_VIEW	BView Hook Functions
Overlapping Siblings	The Interface Kit

P

B_PULSE	Global Variables, Constants, and Defined Types
B_PAGE_DOWN	Global Variables, Constants, and Defined Types
B_PAGE_UP	Global Variables, Constants, and Defined Types
B_PANEL_BACKGROUND_COLOR	Global Variables, Constants, and Defined Types
PaperRect()	BPrintJob
B_PARALLEL_ACCESS	BScreen
Parent()	View Hierarchy Functions
Paste()	BTextView
Pattern Constants	Global Variables, Constants, and Defined Types
pattern	Global Variables, Constants, and Defined Types

Patterns	Drawing
B_PAUSE_KEY	Global Variables, Constants, and Defined Types
The Pen	Drawing
Pen Functions	Graphics State Functions
PenLocation()	Graphics State Functions
PenSize()	Graphics State Functions
Picking Pixels to Stroke and Fill	The Coordinate Space
BPicture	BPicture
BPicture()	BPicture
~BPicture()	BPicture
BPictureButton	BPictureButton
BPictureButton()	BPictureButton
~BPictureButton()	BPictureButton
The Picture Definition	BPicture
Pixel and Coordinate Dimensions	Drawing
B_PLAIN_BORDER	Global Variables, Constants, and Defined Types
Play()	BPicture
PointAt()	BTextView
BPoint	BPoint
BPoint()	BPoint
BPolygon	BPolygon
BPolygon()	BPolygon
~BPolygon()	BPolygon

PopState()	Graphics State Functions
PopUp()	BDragger
BPopUpMenu	BPopUpMenu
BPopUpMenu()	BPopUpMenu
~BPopUpMenu()	BPopUpMenu
Position()	BSlider
B_POSITIVE_HSYNC	BScreen
B_POSITIVE_VSYNC	BScreen
Preparing to Draw	Drawing
PreviousSibling()	View Hierarchy Functions
B_PRIMARY_MOUSE_BUTTO	Input-Related Functions
B_PRIMARY_MOUSE_BUTTON	Global Variables, Constants, and Defined Types
Primitive Drawing Functions	BView Drawing-Related Functions
BPrintJob	BPrintJob
BPrintJob()	BPrintJob
~BPrintJob()	BPrintJob
PrintToStream()	BFont
PrintToStream()	BPoint
PrintToStream()	BPolygon
PrintToStream()	BRect
PrintToStream()	BRegion
print_file_header	Global Variables, Constants, and Defined Types
B_PRINT_KEY	Global Variables, Constants, and Defined Types

PrintableRect()	BPrintJob
PrinterType()	BPrintJob
Printing	BPrintJob
Proportion()	BScrollBar
proportional	Global Functions
ProposeMode()	BScreen
Pulse()	BTextView
Pulse()	BView Hook Functions
PulseRate()	BWindow
B_PULSE_NEEDED	General BView Functions
B_PULSE_NEEDED	Global Variables, Constants, and Defined Types
PushState()	Graphics State Functions

Q

Quit()	BWindow
B_QUIT_ON_WINDOW_CLOSE	BWindow
B_QUIT_REQUESTED	Global Variables, Constants, and Defined Types

---

R

B_RETURN	Global Variables, Constants, and Defined Types
B_RGBA15	Global Variables, Constants, and Defined Types

B_RGBA32	Global Variables, Constants, and Defined Types
B_RGB15	Global Variables, Constants, and Defined Types
B_RGB16	Global Variables, Constants, and Defined Types
B_RGB24	Global Variables, Constants, and Defined Types
B_RGB32	Global Variables, Constants, and Defined Types
BRadioButton	BRadioButton
BRadioButton()	BRadioButton
~BRadioButton()	BRadioButton
ReadBitmap()	BScreen
Recording a Picture	BPicture
RectAt()	BRegion
BRect	BRect
BRect()	BRect
Rectangle Size and Area	BRect
Rectangle Validity	BRect
BRegion	BRegion
BRegion()	BRegion
~BRegion	BRegion
RemoveChild()	BBitmap
RemoveChild()	BWindow
RemoveChild()	View Hierarchy Functions
RemoveFromSubset()	BWindow

RemoveItem()	BListView
RemoveItem()	BMenu
RemoveItem()	BOutlineListView
RemoveItems()	BListView
RemoveItems()	BOutlineListView
RemoveSelf()	View Hierarchy Functions
RemoveShortcut()	BWindow
RemoveTab()	BTabView
ReplaceItem()	BListView
ReplicantAt()	BShelf
ReplicantDeleted()	BShelf
The Replicant Property	BShelf
Reset()	BStatusBar
ResizeBy()	BWindow
ResizeBy()	General BView Functions
ResizeTo()	BWindow
ResizeTo()	General BView Functions
ResizeToPreferred()	BSlider
ResizeToPreferred()	BTextControl
ResizeToPreferred()	BView Hook Functions
Resizing	BTextView
ResizingMode()	General BView Functions
Resizing Modes	Global Variables, Constants, and Defined Types
ResolveSpecifier()	BControl
ResolveSpecifier()	BTextControl

ResolveSpecifier()	BWindow
B_RGB15_BIG	Global Variables, Constants, and Defined Types
B_RGB16_BIG	Global Variables, Constants, and Defined Types
B_RGB24_BIG	Global Variables, Constants, and Defined Types
B_RGB32_BIG	Global Variables, Constants, and Defined Types
B_RGB_16_BIT	Global Variables, Constants, and Defined Types
B_RGB_32_BIT	Global Variables, Constants, and Defined Types
rgb_color	Global Variables, Constants, and Defined Types
B_RGBA15_BIG	Global Variables, Constants, and Defined Types
B_RGBA32_BIG	Global Variables, Constants, and Defined Types
B_RIGHT_ARROW	Global Variables, Constants, and Defined Types
B_RIGHT_COMMAND_KEY	Global Variables, Constants, and Defined Types
B_RIGHT_CONTROL_KEY	Global Variables, Constants, and Defined Types
B_RIGHT_OPTION_KEY	Global Variables, Constants, and Defined Types
B_RIGHT_SHIFT_KEY	Global Variables,

	Constants, and Defined Types
Rotation()	BFont
B_ROUND_C	Graphics State Functions
B_ROUND_JOI	Graphics State Functions
RunArray()	BTextView
run_add_printer_panel()	Global Functions
run_select_printer_panel()	Global Functions

S

ASCII Compatibility	Listening to the User
B_SCROLL	BScreen
B_SPACE	Global Variables, Constants, and Defined Types
Save()	BShelf
SaveLocation()	BShelf
BScreen	BScreen
BScreen()	BScreen
~BScreen()	BScreen
ScreenChanged()	BWindow
The Screen	Drawing
ScreenLocation()	BMenu
ScreenLocation()	BPopUpMenu
Screen Pixels	The Coordinate Space
Screen Spaces	Global Variables, Constants, and Defined

	Types
B_SCREEN_CHANGED	Global Variables, Constants, and Defined Types
screen_id	Global Variables, Constants, and Defined Types
screen_info	Global Variables, Constants, and Defined Types
Scripting and Archival	Scripting and Archival
Scripting and Archival	Scripting and Archival
Scripting Properties	Scripting and Archival
Scripting Suites	Scripting and Archival
Scripting Support	BControl
Scripting Support	BListView
Scripting Support	BMenu
Scripting Support	BShelf
Scripting Support	BTextView
Scripting Support	BWindow
BScrollBar	BScrollBar
BScrollBar()	BScrollBar
~BScrollBar()	BScrollBar
ScrollBar()	BScrollView
Scroll Bar Constants	Global Variables, Constants, and Defined Types

ScrollBar()	Input-Related Functions
Scroll Bar Options	BScrollBar
ScrollBy()	Input-Related Functions
ScrollTo()	BListView
ScrollTo()	Input-Related Functions
ScrollToOffset()	BTextView
ScrollToSelection()	BListView
ScrollToSelection()	BTextView
BScrollView	BScrollView
BScrollView()	BScrollView
~BScrollView()	BScrollView
scroll_bar_info	Global Variables, Constants, and Defined Types
B_SCROLL_KEY	Global Variables, Constants, and Defined Types
B_SCROLL_LOCK	Global Variables, Constants, and Defined Types
Scrolling	Drawing
B_SECONDARY_MOUSE_BU	Input-Related Functions
B_SECONDARY_MOUSE_BUTTON	Global Variables, Constants, and Defined Types
see	BStringView
SelectAll()	BTextView

Select()	BListItem
Select()	BListView
Select()	BTabView
Select()	BTab
Select()	BTextView
Selecting and Invoking Items	BListView
Selection()	BTabView
SelectionChanged()	BListView
SelectionCommand()	BListView
SelectionMessage()	BListView
Selection modes	Drawing
The Selection Property	BListView
The Selection Property	BTextView
SendBehind()	BWindow
separator	Global Functions
SetAlignment()	BMenuField
SetAlignment()	BStringView
SetAlignment()	BTextControl
SetAlignment()	BTextView
SetAllowsDragging()	BShelf
SetAllowsZombies()	BShelf
SetAsyncAutoDestruct()	BPopUpMenu
SetAutoindent()	BTextView
Set()	BPoint
Set()	BRect
Set()	BRegion

SetBarColor()	BSlider
SetBarColor()	BStatusBar
SetBarHeight()	BStatusBar
SetBarThickness()	BSlider
SetBehavior()	BPictureButton
SetBits()	BBitmap
SetBlendingMode()	Graphics State Functions
SetBorder()	BBox
SetBorder()	BMenuBar
SetBorder()	BScrollView
SetBorderHighlighted()	BScrollView
SetCellSize()	BColorControl
SetColorSpace()	BTextView
SetDPMS()	BScreen
SetDefaultButton()	BWindow
SetDesktopColor()	BScreen
SetDirty()	BShelf
SetDisabledOff()	BPictureButton
SetDisabledOn()	BPictureButton
SetDisplayZombies()	BShelf
SetDivider()	BMenuField
SetDivider()	BTextControl
SetDoesUndo()	BTextView
SetDrawingMode()	Graphics State Functions
SetEnabled()	BColorControl
SetEnabled()	BControl

SetEnabled()	BListItem
SetEnabled()	BMenuField
SetEnabled()	BMenuItem
SetEnabled()	BMenu
SetEnabled()	BTab
SetEnabled()	BTextControl
SetEnabledOff()	BPictureButton
SetEnabledOn()	BPictureButton
SetEncoding()	BFont
SetEventMask()	Input-Related Functions
SetExpanded()	BListItem
SetFace()	BFont
SetFamilyAndFace()	BFont
SetFamilyAndStyle()	BFont
SetFeel()	BWindow
SetFlags()	BFont
SetFlags()	BWindow
SetFlags()	General BView Functions
SetFocusTab()	BTabView
SetFontAndColor()	BTextView
SetFont()	BSlider
SetFont()	Graphics State Functions
SetFontSize()	Graphics State Functions
SetHashMarkCount()	BSlider
SetHashMarks()	BSlider
SetHeight()	BListItem

SetHighColor()	Graphics State Functions
SetInvocationMessage()	BListView
SetItemMargins()	BMenu
SetKeyIncrementValue()	BSlider
SetKeyMenuBar()	BWindow
SetLabel()	BBox
SetLabel()	BButton
SetLabel()	BControl
SetLabel()	BMenuField
SetLabel()	BMenuItem
SetLabel()	BTab
SetLabelFromMarked()	BMenu
SetLayout()	BColorControl
SetLeftBottom()	BRect
SetLeftTop()	BRect
SetLimitLabels()	BSlider
SetLineMode()	Graphics State Functions
SetListType()	BListView
SetLook()	BWindow
SetLowColor()	Graphics State Functions
SetMarked()	BMenuItem
SetMaxBytes()	BTextView
SetMaxContentWidth()	BMenu
SetMaxValue()	BStatusBar
SetMode()	BScreen
SetModificationMessage()	BSlider

SetModificationMessage()	BTextControl
SetMouseEventMask()	Input-Related Functions
SetOrientation()	BSlider
SetOrigin()	General BView Functions
SetPenSize()	Graphics State Functions
SetPopUp()	BDragger
SetPosition()	BSlider
SetProportion()	BScrollBar
SetPulseRate()	BWindow
SetRadioMode()	BMenu
SetRange()	BScrollBar
SetResizingMode()	General BView Functions
SetRightBottom()	BRect
SetRightTop()	BRect
SetRotation()	BFont
SetRunArray()	BTextView
SetSaveLocation()	BShelf
SetScale()	Graphics State Functions
SetSelectionMessage()	BListView
SetSettings()	BPrintJob
SetShear()	BFont
SetShortcut()	BAlert
SetShortcut()	BMenuItem
SetSize()	BFont
SetSizeLimits()	BWindow
SetSnoozeAmount()	BSlider

SetSpacing()	BFont
SetSteps()	BScrollBar
SetStylable()	BTextView
SetStyle()	BSlider
SetTabHeight()	BTabView
SetTabWidth()	BTabView
SetTabWidth()	BTextView
SetTarget()	BScrollBar
SetTargetForItems()	BMenu
SetText()	BStatusBar
SetText()	BStringItem
SetText()	BStringView
SetText()	BTextControl
SetText()	BTextView
SetTextRect()	BTextView
SetTitle()	BWindow
SetToNext	BScreen
SetTrailingText()	BStatusBar
SetTrigger()	BMenuItem
SetTriggersEnabled()	BMenu
SetType()	BWindow
SetTypeEnforced()	BShelf
SetValue()	BColorControl
SetValue()	BControl
SetValue()	BRadioButton
SetValue()	BScrollBar

SetValue()	BSlider
SetView()	BTab
SetViewBitmap()	BView Drawing-Related Functions
SetViewColor()	Graphics State Functions
SetViewCursor()	General BView Functions
SetViewOverlay()	BView Drawing-Related Functions
SetWidth()	BListItem
SetWindowAlignment()	BWindow
SetWordWrap()	BTextView
SetWorkspaces()	BWindow
SetZoomLimits()	BWindow
set_menu_info()	Global Functions
set_screen_space()	Global Functions
set_scroll_bar_info()	Global Functions
set_workspace_count()	Global Functions
Setting Navigation Flags	Listening to the User
Setting Up a Print Job	BPrintJob
Setting Up the Page Layout	BPrintJob
Settings()	BPrintJob
BShape	BShape
BShape()	BShape
~BShape()	BShape
BShapeIterator	BShapeIterator
BShapeIterator()	BShapeIterator

~BShapeIterator()	BShapeIterator
Shear()	BFont
BShelf	BShelf
BShelf()	BShelf
~BShelf()	BShelf
"Shelf"	Scripting and Archival
B_SHIFT_KEY	Global Variables, Constants, and Defined Types
B_SHIFT_TABLE	Global Variables, Constants, and Defined Types
Shortcut()	BAlert
Shortcut()	BMenuItem
Shortcuts and Menu Items	BTextView
Shortcuts and Triggers	BMenuItem
ShowAllDraggers()	BDragger
Show()	BMenuBar
Show()	BMenu
Show()	BWindow
Show()	General BView Functions
The Signature Property	BShelf
B_SINGLE_SELECTION_LIST	Global Variables, Constants, and Defined Types
Size()	BFont
BSlider	BSlider

BSlider()	BSlider
~BSlider()	BSlider
BSlider: Customizing your Slider	BSlider: Customizing your Slider
BSlider: Customizing your Slider	BSlider: Customizing your Slider
SnoozeAmount()	BSlider
SortItems()	BListView
SortItemsUnder()	BOutlineListView
Spacing()	BFont
The Spool File	BPrintJob
SpoolPage()	BPrintJob
B_SQUARE_C	Graphics State Functions
B_SQUARE_JO	Graphics State Functions
Static Functions	BBitmap
Static Functions	BColorControl
Static Functions	BControl
Static Functions	BDragger
Static Functions	BListView
Static Functions	BMenuBar
Static Functions	BMenuField
Static Functions	BMenu
Static Functions	BOutlineListView
Static Functions	BPopUpMenu
Static Functions	BRadioButton
Static Functions	BScrollBar

Static Functions	BScrollView
Static Functions	BShape
Static Functions	BShelf
Static Functions	BStatusBar
Static Functions	BStringItem
Static Functions	BStringView
Static Functions	BTabView
Static Functions	BTab
Static Functions	BTextControl
Static Functions	BTextView
Static Functions	BWindow
Static Functions	General BView Functions
BStatusBar	BStatusBar
BStatusBar()	BStatusBar
~BStatusBar()	BStatusBar
BStringItem	BStringItem
BStringItem()	BStringItem
~BStringItem()	BStringItem
String Truncation Constants	Global Variables, Constants, and Defined Types
BStringView	BStringView
BStringView()	BStringView
~BStringView()	BStringView
StringWidth()	BFont
StringWidth()	Graphics State Functions

B_STRING_SPACING	Global Variables, Constants, and Defined Types
StrokeArc()	BView Drawing-Related Functions
StrokeBezier()	BView Drawing-Related Functions
StrokeEllipse()	BView Drawing-Related Functions
StrokeLine()	BView Drawing-Related Functions
StrokePolygon()	BView Drawing-Related Functions
StrokeRect()	BView Drawing-Related Functions
StrokeRoundRect()	BView Drawing-Related Functions
StrokeShape()	BView Drawing-Related Functions
StrokeTriangle()	BView Drawing-Related Functions
Stroking Curved Lines	The Coordinate Space
Stroking Thick Lines	The Coordinate Space
Stroking Thin Lines	The Coordinate Space
Style()	BSlider
SubmenuAt()	BMenu
Submenu()	BMenuItem
B_SUBPIXEL_PRECISE	General BView Functions

The Suites Property	BShelf
Superitem()	BMenu
Superitem()	BOutlineListView
Supermenu()	BMenu
SwapItems()	BListView
Sync()	BView Drawing-Related Functions
Sync()	BWindow
B_SYNC_ON_GREEN	BScreen
Synchronizing a List Item with its List View	BListItem
System Fonts	BFont
System Fonts	Global Variables, Constants, and Defined Types
system_colors()	Global Functions

T

B_TAB	Global Variables, Constants, and Defined Types
TabAt()	BTabView
BTab	BTab
BTab()	BTab
~BTab()	BTab
TabFrame()	BTabView

TabHeight()	BTabView
BTabView	BTabView
BTabView()	BTabView
~BTabView()	BTabView
TabWidth()	BTabView
TabWidth()	BTextView
B_TAB_ANY	BTab
B_TAB_FIRST	BTab
B_TAB_FRONT	BTab
Target()	BScrollBar
TargetedByScrollView()	BListView
TargetedByScrollView()	BView Hook Functions
B_TERTIARY_MOUSE_BUTT	Input-Related Functions
B_TERTIARY_MOUSE_BUTTON	Global Variables, Constants, and Defined Types
Text()	BStatusBar
Text()	BStringItem
Text()	BStringView
Text()	BTextControl
Text()	BTextView
BTextControl	BTextControl
BTextControl()	BTextControl
~BTextControl()	BTextControl
Text Functions	Graphics State Functions
TextHeight()	BTextView

TextLength()	BTextView
The Text Property	BTextView
TextRect()	BTextView
BTextView and BScrollBars	BTextView
TextView()	BAlert
TextView()	BTextControl
BTextView	BTextView
BTextView()	BTextView
~BTextView()	BTextView
text_run	Global Variables, Constants, and Defined Types
text_run_array	Global Variables, Constants, and Defined Types
The text_run_array Property	BTextView
ThumbFrame()	BSlider
B_TIMING_INTERLACED	BScreen
tint_color() constants	Global Variables, Constants, and Defined Types
"Title"	BWindow
B_TITLED_WINDOW	BWindow
B_TITLED_WINDOW	Global Variables, Constants, and Defined Types
B_TITLED_WINDOW_LOOK	BWindow

Track()	BMenu
B_TRACK_RECT_CORNER	Input-Related Functions
B_TRACK_WHOLE_RECT	Input-Related Functions
Tracking the Keyboard	Listening to the User
TrailingLabel()	BStatusBar
TrailingText()	BStatusBar
Transparency	BBitmap
Transparency Constants	Global Variables, Constants, and Defined Types
Transparency modes	Drawing
B_TRANSPARENT_MAGIC_CMAP8	Drawing
B_TRANSPARENT_MAGIC_RGBA15	Drawing
B_TRANSPARENT_MAGIC_RGBA32	Drawing
B_TRIANGLE_THUMB	BSlider
Trigger()	BMenuItem
triggers_always_shown	Global Functions
TruncateLabel()	BMenuItem
TruncateString()	BFont
TruncateString()	Graphics State Functions
B_TRUNCATE_BEGINNING	Global Variables, Constants, and Defined Types
B_TRUNCATE_END	Global Variables, Constants, and Defined Types
B_TRUNCATE_MIDDLE	Global Variables,

	Constants, and Defined Types
B_TRUNCATE_SMART	Global Variables, Constants, and Defined Types
tuned_font_info	BFont
tuned_font_info	Global Variables, Constants, and Defined Types
Two-Dimensional Coordinates	The Coordinate Space
Type()	BWindow

U

UTF-8 and the BeOS	Listening to the User
UTF-8	Listening to the User
B_UVLA32	Global Variables, Constants, and Defined Types
B_UVL24	Global Variables, Constants, and Defined Types
B_UVL32	Global Variables, Constants, and Defined Types
Undo()	BTextView
UndoState()	BTextView
undo_state	BTextView
UnflattenRunArray()	BTextView
unicode_block	BFont
B_UNICODE_UTF8	Global Variables,

	Constants, and Defined Types
Unlock()	BBitmap
B_UNTYPED_WINDOW	BWindow
B_UP_ARROW	Global Variables, Constants, and Defined Types
Update()	BListItem
Update()	BStatusBar
Update()	BStringItem
UpdateIfNeeded()	BWindow
The Update Mechanism	BScrollBar
The Update Mechanism	Drawing
UpdateText()	BSlider
update_font_families()	Global Functions
Updating the List	BListView
UseFillColor()	BSlider
The User Interface	Listening to the User
Using a BFont Object	BFont
Using a CityItem Object	BListItem: Creating a Custom List Item
Using a View to Draw into a Bitmap	BBitmap
Using BSliders	BSlider
B_UTF8_CLOSE_QUOTE	Global Variables, Constants, and Defined Types
B_UTF8_COPYRIGHT	Global Variables, Constants, and Defined Types
B_UTF8_ELLIPSIS	Global Variables,

	Constants, and Defined Types
B_UTF8_HIROSHI	Global Variables, Constants, and Defined Types
B_UTF8_OPEN_QUOTE	Global Variables, Constants, and Defined Types
B_UTF8_REGISTERED	Global Variables, Constants, and Defined Types
B_UTF8_SMILING_FACE	Global Variables, Constants, and Defined Types
B_UTF8_TRADEMARK	Global Variables, Constants, and Defined Types

V

B_VERTICAL	Global Variables, Constants, and Defined Types
B_V_SCROLL_BAR_WIDTH	Global Variables, Constants, and Defined Types
Value and Range	BScrollBar
ValueAsColor()	BColorControl
Value()	BControl
Value()	BScrollBar
Value()	BSlider
ValueChanged()	BScrollBar
ValueForPoint()	BSlider
The Value Property	BControl
B_VALUE_CHANGED	Global Variables, Constants, and Defined Types

View()	BTab
BView	BView
BView	BView
"View"	BWindow
The View Color	Drawing
ViewColor()	Graphics State Functions
BView Drawing-Related Functions	BView Drawing-Related Functions
BView Drawing-Related Functions	BView Drawing-Related Functions
View Flags	Global Variables, Constants, and Defined Types
BView()	General BView Functions
~BView()	General BView Functions
View Geometry	Drawing
View Hierarchy Functions	View Hierarchy Functions
View Hierarchy Functions	View Hierarchy Functions
The View Hierarchy	The Interface Kit
BView Hook Functions	BView Hook Functions
BView Hook Functions	BView Hook Functions
BView Introduction	BView Introduction
BView Introduction	BView Introduction
BView Objects	The Interface Kit
The View Property	BShelf
"View"	Scripting and Archival
The View Structure	BView Introduction

View's frame rectangle	Scripting and Archival
B_VIEW_MOVED	Global Variables, Constants, and Defined Types
B_VIEW_RESIZED	Global Variables, Constants, and Defined Types
Views and the Server	Drawing

W

WaitForRetrace()	BScreen
Width()	BListItem
Width()	BRect
B_WIDTH_AS_USUAL	Global Variables, Constants, and Defined Types
B_WIDTH_AS_USUAL	BTabView
B_WIDTH_FROM_LABEL	Global Variables, Constants, and Defined Types
B_WIDTH_FROM_LABEL	BTabView
B_WIDTH_FROM_WIDEST	Global Variables, Constants, and Defined Types
B_WIDTH_FROM_WIDEST	BTabView
B_WILL_ACCEPT_FIRST_CLICK	BWindow
B_WILL_ACCEPT_FIRST_CLICK	Global Variables, Constants, and Defined Types
B_WILL_DRAW	General BView Functions
B_WILL_DRAW	Global Variables, Constants, and Defined Types

B_WILL_FLOAT	Global Variables, Constants, and Defined Types
WindowActivated()	BControl
WindowActivated()	BListView
WindowActivated()	BMenuField
WindowActivated()	BTabView
WindowActivated()	BTextControl
WindowActivated()	BTextView
WindowActivated()	BView Hook Functions
WindowActivated()	BWindow
BWindow	BWindow
BWindow()	BWindow
~BWindow()	BWindow
Window Flags	Global Variables, Constants, and Defined Types
Window Flags.	BWindow
Window()	General BView Functions
BWindow Objects	The Interface Kit
Window Workspaces	BWindow
B_WINDOW_ACTIVATED	Global Variables, Constants, and Defined Types
window_feel	BWindow
window_look	BWindow
B_WINDOW_MOVE_BY	BWindow
B_WINDOW_MOVE_TO	BWindow
B_WINDOW_MOVED	Global Variables, Constants, and Defined Types

B_WINDOW_RESIZED	Global Variables, Constants, and Defined Types
B_WINDOW_TAB_COLOR	Global Variables, Constants, and Defined Types
window_type	BWindow
window_type Constants	Global Variables, Constants, and Defined Types
window_type	Global Variables, Constants, and Defined Types
WorkspaceActivated()	BWindow
Workspace Constants	Global Variables, Constants, and Defined Types
B_WORKSPACE_ACTIVATED	Global Variables, Constants, and Defined Types
"Workspaces"	BWindow
WorkspacesChanged()	BWindow
B_WORKSPACES_CHANGED	Global Variables, Constants, and Defined Types

---

Y

B_YCbCr411	Global Variables, Constants, and Defined Types
B_YCbCr420	Global Variables, Constants, and Defined Types
B_YCbCr422	Global Variables, Constants, and Defined Types
B_YCbCr444	Global Variables,

	Constants, and Defined Types
B_YUV12	Global Variables, Constants, and Defined Types
B_YUV411	Global Variables, Constants, and Defined Types
B_YUV420	Global Variables, Constants, and Defined Types
B_YUV422	Global Variables, Constants, and Defined Types
B_YUV444	Global Variables, Constants, and Defined Types
B_YUV9	Global Variables, Constants, and Defined Types

Z

B_ZOOM	Global Variables, Constants, and Defined Types
Zoom()	BWindow

|

||	BRect

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFont

Derived from: *none*
Declared in: be/interface/Font.h
Library: libbe.so
Summary: *more...*

A BFont object records a set of font attributes, such as the font's family, style, size, and so on. You can set most of these attributes to modify the font and then use the object to set the font of a BView. A BView's font determines how the characters that it draws (with the **DrawString()** and **DrawChar()** functions) will be rendered.

A BFont object can perform calculations based on the metrics of the particular font it represents. For example, it can tell you how much screen real estate it needs to display a given line of text.

To find which fonts are currently installed on the system, call **get_font_family()** and **get_font_style()**.

Using a BFont Object

A BFont object by itself doesn't do anything. To be able to draw characters in the font, the object must be passed to BView's **SetFont()** function (or BTextView's **SetFontAndColor()**).

A BFont object is always a full representation of a font; all attributes are always set. However, you can choose which of these attributes will modify a BView's current font by passing a mask to **SetFont()** (or **SetFontAndColor()**). For example, this code sets only the font shear and spacing:

```
BFont font;
font.SetShear(60.0);
font.SetSpacing(B_CHAR_SPACING);
myView->SetFont(&font, B_FONT_SHEAR | B_FONT_SPACING);
```

Alternatively, the BView's font could have been modified and reset as follows:

```
BFont font;
myView->GetFont(&font);
font.SetShear(60.0);
font.SetSpacing(B_CHAR_SPACING);
myView->SetFont(&font);
```

Notice that we had to explicitly reset the view's font (through **SetFont()**) after changing the font's attributes.

System Fonts

The Interface Kit constructs three BFont objects (plain, bold, and fixed)for each application when the application starts up. The values of these fonts are set by the user through the FontPanel preferences application. You can get to these objects through global pointers :

const BFont * **be_plain_font**
The font that's used to display most gadgets in the user interface, such as check box labels and menu items. All BControl objects use this font.

const BFont * **be_bold_font**
The font that's used to display window titles and BBox labels.

const BFont * **be_fixed_font**
The font that's used to display fixed-width characters.

The global fonts are `const` objects that can't be modified by your application, and aren't updated by the system, even if the user changes their definitions while your app is running. The new values take effect the next time your application is launched.

To use a system font in a view, simply call **SetFont()**:

```
myView->SetFont(be_bold_font);
```

If you want to modify some attributes of a system font, you have to make a copy of it first (and modify the copy):

```
BFont font(be_bold_font);
font.SetSize(13.0);
myView->SetFont(&font);
```

Applications should respect the user's choices and base all font choices on these three system fonts, rather than hard-code font names into the application. You should not try to predict the fonts that will be installed on the user's machine.

Constructor

BFont()

```
BFont(const BFont &font)
BFont(const BFont *font)
BFont(void)
```

Initializes the new BFont object as a copy of another *font*. If no font is specified, **be_plain_font** is used.

The system BFont objects, including **be_plain_font**, are initialized only when you create a BApplication object for your application. Therefore, the default settings of BFont objects constructed before the BApplication object will be invalid.

See also: **BView::SetFont()**, **BTextView::SetFontAndColor()**

Member Functions

Blocks()

```
unicode_block Blocks(void) const
```

Returns a unicode_block object that identifies which Unicode blocks the font supports. You can then use the **`unicode_block::Includes()`** function to determine if specific blocks are supported.

BoundingBox()

```
BRect BoundingBox(void) const
```

Returns a BRect that can enclose the entire font in its current style and size.

CountTuned() see GetTunedInfo()

Direction() , font_direction

```
font_direction Direction(void) const
enum font_direction {}
```

`Direction()` returns a `font_direction` constant that describes the direction in which the object's text is meant to be read:

`B_FONT_LEFT_TO_RIGHT`
`B_FONT_RIGHT_TO_LEFT`

This is an inherent property of the font and cannot be set.

The direction of the font affects the direction in which **`DrawString()`** draws the characters in a string, but not the direction in which it moves the pen.

See also: **`BView::DrawString()`**

Encoding() see SetEncoding()

Face() see SetFace()

FamilyAndStyle() see SetFamilyAndStyle()

FileFormat()

font_file_format **FileFormat(**void**)** const

Returns the file format of the font (ie, whether it's a PostScript or TrueType font).

Flags() see SetFlags()

GetBoundingBoxesAsGlyphs() , GetBoundingBoxesAsString() , GetBoundingBoxesForStrings()

void **GetBoundingBoxesAsGlyphs(**const char *charArray*[], int32 *numChars*,
font_metric_mode *mode*, BRect *boundingBoxArray*[]**)** const

void **GetBoundingBoxesAsString(**const char *string*[], int32 *numChars*,
font_metric_mode *mode*,
escapement_delta **delta*,
BRect *boundingBoxArray*[]**)** const

void **GetBoundingBoxesForStrings(**const char **stringArray*[], int32 *numStrings*,
font_metric_mode *mode*,
escapement_delta **deltas*[],
BRect *boundingBoxArray*[]**)** const

`GetBoundingBoxesAsGlyphs()` returns an array of BRect objects indicating the bounding rectangles of all the characters in an array. Each BRect returned corresponds to one character's glyph.

`GetBoundingBoxesAsString()` returns an array of BRect objects indicating the bounding rectangles of each character in a *string*.

`GetBoundingBoxesForStrings()` returns an array of BRect objects indicating the bounding rectangles for an array of strings, one BRect per string. These rectangles enclose the entire string they represent.

In all cases, the *mode* indicates whether the rectangles should be returned in the screen's metric (**`B_SCREEN_METRIC`**), or in printing metrics (**`B_PRINTING_METRIC`**).

The *delta* argument for `GetBoundingBoxesAsString()` and the *deltas* argument for `GetBoundingBoxesForStrings()` indicate escapement deltas that should be applied when making the bounding box calculations. This lets you indicate that the characters should be closer together or further apart than normal, for example.

GetEscapements() , escapement_delta , GetEdges() , edge_info

void **GetEscapements(**const char *charArray*[], int32 *numChars*,

```
    float escapementArray[]) const
void GetEscapements(const char charArray[], int32 numChars,
    escapement_delta *delta, float escapementArray[]) const
void GetEscapements(const char charArray[], int32 numChars,
    escapement_delta *delta, BPoint escapementArray[]) const
void GetEscapements(const char charArray[], int32 numChars,
    escapement_delta *delta, BPoint escapementArray[],
    BPoint offsetArray[]) const
```

```
struct escapement_delta {
    float nonspace;
    float space;
    }
```

```
void GetEdges(const char charArray[], int32 numChars,
    edge_info edgeArray[]) const
```

```
struct edge_info {
    float left;
    float right;
    }
```

These two functions provide the information required to precisely position characters on the screen or printed page. For each character passed in the *charArray*, they write information about the horizontal dimension of the character into the *escapementArray* or the *edgeArray*. Both functions provide this information in "escapement units" that yield standard coordinate units (72.0 per inch) when multiplied by the font size.

`GetEscapements()` and `GetEdges()` expect the character array they're passed to contain at least *numChar* characters; neither function checks the *charArray* for a null terminator. Because the array may hold multibyte characters (in `B_UNICODE_UTF8` encoding), the number of bytes in the array may be greater than the number of characters specified. The *escapementArray* and *edgeArray* should be long enough to hold an output value for every input character.

You can optionally request that escapement information be returned as an array of BPoint objects.

Escapements

A character's escapement measures the amount of horizontal space it requires. It includes the space needed to display the character itself, plus some extra room on the left and right edges to separate the character from its neighbors. The illustration below shows the approximate escapements for the letters 'l' and 'p'; the escapement for each character is the distance between the vertical lines:

`GetEscapements()` measures the same space that functions such as **`StringWidth()`** and BTextView's **`LineWidth()`** do, but it measures each character individually and records its width in per-point-size escapement units. To translate the escapement value to the width of the character, you must multiply by the point size of the font:

```
float width = escapementArray[i] * font.Size();
```

Because of rounding errors, there may be some difference between the value returned by **`StringWidth()`** and the width calculated from the individual escapements of the characters in the string.

The versions of `GetEscapements()` that use BPoints for the escapement value use the BPoint *escapementArray* to indicate a vector by which the pen is moved after drawing a character (this lets the escapement indicate both an X and a Y adjustment; the Y might need to be adjusted if the font is rotated, for example). The *offsetArray* is applied by the dynamic spacing in order to improve the relative position of the character's width with relation to another character, without altering the width.

The escapement value is scalable if the spacing mode of the font is **`B_CHAR_SPACING`**. In other words, given **`B_CHAR_SPACING`** and the same set of font characteristics, `GetEscapements()` will report the same measurement for a character regardless of the font size. You can cache one value per character and use it for all font sizes. For the other spacing modes, the reported escapement depends on the font size and therefore can't be scaled.

For most spacing modes, a character has a constant escapement in all contexts; it depends only on the font. However, for **`B_STRING_SPACING`**, each character's escapement is also contextually dependent on the string it's in. To find the escapement of a character within a particular string, you must pass the entire string in the input *charArray*.

In the **`B_BITMAP_SPACING`** and **`B_FIXED_SPACING`** modes, all characters have integral widths (without a fractional part). For these modes, multiplying an escapement by the font size should yield an integral value. In **`B_FIXED_SPACING`** mode, all characters have the same escapement.

If a *delta* argument is provided, **`GetEscapements()`** will adjust the escapements it reports so that, after multiplying by the font size, the character widths will include the specified increments. An `escapement_delta` structure contains two values:

float **nonspace**
: The amount to add to the width of each character with a visible glyph.

float **space**
The amount to add to each whitespace character (characters like **B_TAB** and **B_SPACE** with an escapement but no visible glyph).

A similar argument can be passed to BView's **DrawString()** to adjust the spacing of the characters as they're drawn.

Edges

Edge values measure how far a character outline is inset from its left and right escapement boundaries. `GetEdges()` places the edge values into an array of **edge_info** structures. Each structure has a `left` and a `right` data member, as follows:

```
typedef struct {
   float left;
   float right;
} edge_info
```

Edge values, like escapements, are stated in per-point-size units that need to be multiplied by the font size.

The illustration below shows typical character edges. As in the illustration above, the solid vertical lines mark escapement boundaries. The dotted lines mark off the part of each escapement that's an edge, the distance between the character outline and the escapement boundary:

This is the normal case. The left edge is a positive value measured rightward from the left escapement boundary. The right edge is a negative value measured leftward from the right escapement boundary.

However, if the characters of a font overlap, the left edge can be a negative value and the right edge can be positive. This is illustrated below:

Note that the italic '*l*' extends beyond its escapement to the right, and that the '*p*' begins before its escapement to the left. In this case, instead of separating the adjacent characters, the edges determine how much they overlap.

Edge values are specific to each character and depend on nothing but the character and the font. They don't take into account any contextual information; for example, the right edge for italic '*l*' would be the same no matter what letter followed. Edge values therefore aren't sufficient to decide how character pairs can be kerned. Kerning is contextually dependent on the combination of two particular characters.

See also: **StringWidth()**, **SetSpacing()**

GetFamilyAndStyle() see SetFamilyAndStyle()

GetGlyphShapes()

```
void GetGlyphShapes(const char charArray[], int32 numChars,
    BShape *glyphShapeArray[]) const
```

Given an array of characters, *charArray*, which contains *numChars* characters, and an array of BShape objects, *glyphShapeArray*, this function makes each element in the *glyphShapeArray* describe the shape of the corresponding glyph in the *charArray*.

This lets you create BShape objects in the shape of the outline of a font. You can then manipulate these BShapes to do interesting text effects.

The *glyphShapeArray* must contain already-allocated BShape objects. They will be cleared by this function before the glyphs' shapes are constructed into them, but the objects must already exist.

Fonts are drawn one pixel above the logical baseline; this affects BShape objects derived from fonts, too. The fonts are also one pixel above the baseline in the BShapes this function returns. If you want to apply a transform to these shapes, be sure to remove the offset before applying the transform, then add the offset back to the points before drawing the shape, or you won't get the expected results.

GetHasGlyphs()

```
void GetHasGlyphs(const char charArray[], int32 numChars,
    bool hasArray[]) const
```

Given an array of characters in *charArray* (of which there are *numChars* members), this function fills out the array of booleans specified by *hasArray* such that each entry in *hasArray* is **true** if the corresponding character in *charArray* has a glyph in the font, and **false** if the character doesn't have a glyph.

This way, you can determine if you can use one or more characters without the user seeing "no glyph" symbols.

GetHeight() , font_height

```
void GetHeight(font_height *height) const
```

```
struct font_height {
    float ascent;
    float descent;
    float leading;
    }
```

GetHeight() writes the three components that determine the height of the font into the structure that the *height* argument refers to. A **font_height** structure has the following fields:

float **ascent**
How far characters can ascend above the baseline.

float **descent**
How far characters can descend below the baseline.

float **leading**
How much space separates lines (the distance between the descent of the line above and the ascent of the line below).

If you need to round the font height, or any of its components, to an integral value (to figure the spacing between lines of text on-screen, for example), you should always round them up to reduce the amount of vertical character overlap.

See also: `BView::GetFontHeight()`

GetStringWidths() see StringWidth()

GetTruncatedStrings() , TruncateString()

void **GetTruncatedStrings(**const char **inputStringArray*[], int32 *numStrings*, uint32 *mode*, float *maxWidth*, char **truncatedStringArray*[]**)** const void **GetTruncatedStrings(**const char **inputStringArray*[], int32 *numStrings*, uint32 *mode*, float *maxWidth*, BString *truncatedStringArray*[]**)** const
void **TruncateString(**BString **inOutString*, uint32 *mode*, float *maxWidth***)** const

`GetTruncatedStrings()` Ttruncates a set of strings so that each one (and an ellipsis to show where the string was cut) will fit into the *maxWidth* horizontal space. This function is useful for shortening long strings that are displayed to the user—for showing path names in a list, for example.

The *numStrings* argument states how many strings in the *inputStringArray* should be shortened. The *mode* argument states where the string should be cut. It can be:

`B_TRUNCATE_BEGINNING`	Cut from the beginning of the string until it fits within the specified width.
`B_TRUNCATE_MIDDLE`	Cut from the middle of the string.
`B_TRUNCATE_END`	Cut from the end of the string.
`B_TRUNCATE_SMART`	Cut anywhere, but do so intelligently, so that all the strings remain different after being cut. For example, if a set of similar path names are passed in the *inputStringArray*, this mode would attempt to cut from the identical parts of the path names and preserve the parts that are different. This mode also pays attention to word boundaries,

	separators, punctuation, and the like. However, it's not implemented for the current release.

Each output string is written to the *truncatedStringArray*—into memory that the caller must provide—at an index that matches the index of the full string in the *inputStringArray*. The *truncatedStringArray* is a list of pointers to string buffers. Each buffer should be allocated separately and should be at least 3 bytes longer than the matching input string. The 3 bytes allow for the worst-case scenario: **GetTruncatedStrings()** cuts a one-byte character from the input string and replaces it with an ellipsis character, which takes three bytes in UTF-8 encoding, for a net gain of 2 bytes. It then adds a null terminator for the third byte.

TruncateString() truncates the BString *inOutString* to be no longer than the width specified by *maxWidth*, using the given truncation *mode*.

The output strings are null-terminated. The input strings should likewise be null-terminated.

See also: **StringWidth()**

GetTunedInfo() , CountTuned() , tuned_font_info

```
void GetTunedInfo(int32 *index, tuned_font_info *info) const
```

```
int32 CountTuned(void) const
```

```
struct tuned_font_info {
    float size;
    float shear;
    float rotation;
    uint32 flags;
    uint16 face;
    }
```

These functions are used to get information about fonts that have been "tuned" to look good when displayed on-screen. A tuned font is a set of character bitmaps, originally produced from the standard outline font and then modified so that the characters are well proportioned and spaced when displayed at the low resolution of the screen (1 pixel per point).

Because it's a bitmap font, a tuned font captures a specific configuration of font attributes, including size, style, shear, and rotation. A tuned font is a counterpart to an outline font with the same settings. If a BView's current font has a tuned counterpart, **DrawString()** automatically chooses it when drawing on-screen. Tuned fonts are not used for printing.

CountTuned() returns how many tuned fonts there are for the family and style represented by the BFont object. **GetTunedInfo()** writes information about the tuned font at *index* into the structure the *info* argument refers to. Indices begin at 0 and count only tuned fonts for the BFont's family and style.

With this information, you can set the BFont to values that match those of a tuned font. When a BView draws to the screen, it picks a tuned font if there's one that corresponds to its current font in all respects.

See also: `get_font_family()`

IsFixed()

```
bool IsFixed(void) const
```

Returns **true** if the font is fixed; **false** otherwise.

IsFullAndHalfFixed()

```
bool IsFullAndHalfFixed(void) const
```

This function is not yet supported.

PrintToStream()

```
void PrintToStream(void) const
```

Writes the following information about the font to the standard output (on a single line):

- *family*
- *style*
- *size* (in points)
- *shear* (in degrees)
- *rotation* (in degrees)
- *ascent*
- *descent*
- *leading*

Rotation() see SetRotation()

SetEncoding() , Encoding()

```
void SetEncoding(uint8 encoding)
uint8 Encoding(void) const
```

These functions set and return the encoding that maps character values to characters. The following encodings are supported:

B_UNICODE_UTF8 (UTF-8)	B_ISO_8859_6 (Latin/Arabic)
B_ISO_8859_1 (Latin 1)	B_ISO_8859_7 (Latin/Greek)
B_ISO_8859_2 (Latin 2)	B_ISO_8859_8 (Latin/Hebrew)
B_ISO_8859_3 (Latin 3)	B_ISO_8859_9 (Latin 5)
B_ISO_8859_4 (Latin 4)	B_ISO_8859_10 (Latin 6)
B_ISO_8859_5 (Latin/Cyrillic)	B_MACINTOSH_ROMAN

UTF-8 is an 8-bit encoding for Unicode and is part of the Unicode standard. It matches ASCII values for all 7-bit character codes, but uses multibyte characters for values over 127. The other encodings take only a single byte to represent a character; they therefore necessarily encompass a far smaller set of characters. Most of them represent standards in the ISO/IEC 8859 family of character codes that extend the ASCII set. B_MACINTOSH_ROMAN stands for the standard encoding used by the Mac OS.

The encoding affects both input and output functions of the BView. It determines how DrawString() interprets the character values it's passed and also how KeyDown() encodes character values for the keys the user pressed.

UTF-8 is the preferred encoding and the one that's most compatible with objects defined in the software kits. For example, a BTextView expects all text it takes from the clipboard or from a dragged and dropped message to be UTF-8 encoded. If it isn't, the results are not defined. The more that applications stick with UTF-8 encoding, the more freely they'll be able to exchange data.

See also: "Character Encoding" on page48 of this chapter, convert_to_utf8(), **BView::DrawString()**, **BView::KeyDown()**

SetFace() , Face()

void **SetFace(**uint16 *face***)**
uint16 **Face(**void**)** const

These functions set and return a mask that record secondary characteristics of the font, such as whether characters are underlined or drawn in outline. The values that form the *face* mask are:

B_ITALIC_FACE
Characters are drawn italicized.

B_UNDERSCORE_FACE
Characters are drawn underlined.

B_NEGATIVE_FACE
Characters are drawn in the low color, while the background is drawn in the high color.

B_OUTLINED_FACE
Characters are drawn hollow, with a line around their border, but unfilled.

B_STRIKEOUT_FACE
Characters are drawn "struck-out," with a line drawn horizontally through the middle.

B_BOLD_FACE
Characters are drawn in boldface.

B_REGULAR_FACE
Characters are drawn normally.

SetFamilyAndFace()

void **SetFamilyAndFace(**const font_family family, uint16 *face***)**

Sets the family and face of the font. The *family* passed to this function must be one of the families enumerated by the `get_font_family()` global function and *face* must be a combination of the face values described under `SetFace()` on page125. If the *family* is `NULL`, `SetFamilyAndFace()` sets only the face.

SetFamilyAndStyle() , GetFamilyAndStyle() , FamilyAndStyle() , font_family , font_style

void **SetFamilyAndStyle(**const font_family *family*, const font_style *style***)** void **SetFamilyAndStyle(**uint32 *code***)**
void **GetFamilyAndStyle(**font_family **family*, font_style **style***)** const
uint32 **FamilyAndStyle(**void**)** const
typedef char **font_family[B_FONT_FAMILY_LENGTH + 1]**
typedef char **font_style[B_FONT_STYLE_LENGTH + 1]**

`SetFamilyAndStyle()` sets the family and style of the font. The *family* passed to this function must be one of the families enumerated by the `get_font_family()` global function and *style* must be one of the styles associated with that family, as reported by `get_font_style()`. If the *family* is `NULL`, `SetFamilyAndStyle()` sets only the style; if *style* is `NULL`, it sets only the family.

`GetFamilyAndStyle()` writes the names of the current family and style into the **font_value** and `font_style` variables provided.

Internally, the BFont class encodes each family and style combination as a unique integer. `FamilyAndStyle()` returns that code, which can then be passed to `SetFamilyAndStyle()` to set another BFont object. The integer code is not persistent; its meaning may change when the list of installed fonts changes and when the machine is rebooted.

See also: `get_font_family()`

SetFlags() , Flags()

void **SetFlags(**uint32 *flags***)**
uint32 **Flags(**void**)** const

These functions set and return a mask that records various behaviors of the font. There are two flags: **B_DISABLE_ANTIALIASING**, which turns off all antialiasing for characters displayed in the font, and **B_FORCE_ANTIALIASING**, which forces all font rendering to be anti-aliased. The default mask has antialiasing turned on.

SetRotation() , Rotation()

void **SetRotation(**float *rotation***)**
float **Rotation(**void**)** const

These functions set and return the rotation of the baseline for characters displayed in the font. The baseline rotates counterclockwise from an axis on the left side of the character. The default (horizontal) baseline is at 0 °. For example, this code

```
BFont font;
font.SetRotation(45.0);
myView->SetFont(&font, B_FONT_ROTATION);
myView->DrawString("to the northeast");
```

would draw a string that extended upwards and to the right.

Rotation is not supported by some Interface Kit classes, including BTextView.

SetShear() , Shear()

void **SetShear(**float *shear***)**
float **Shear(**void**)** const

These functions set and return the angle at which characters are drawn relative to the baseline. The default (perpendicular) shear for all font styles, including oblique and italic ones, is 90.0 °. The shear is measured counterclockwise and can be adjusted within the range 45.0 ° (slanted to the right) through 135.0 ° (slanted to the left). If the *shear* passed falls outside this range, it will be adjusted to the closest value within range.

SetSize() , Size()

void **SetSize(**float *size***)**
float **Size(**void**)** const

These functions set and return the size of the font in points. Valid sizes range from less than 1.0 point through 10,000 points.

See also: `BView::SetSize()`

SetSpacing() , Spacing()

void **SetSpacing(**uint8 *spacing***)**
uint8 **Spacing(**void**)** const

These functions set and return the mode that determines how characters are horizontally spaced relative to each other when they're drawn. The mode also affects the width or "escapement" of each character as reported by `GetEscapements()`.

There are four *spacing* modes:

`B_CHAR_SPACING`
Positions each character according to its own inherent width, without adjustment. This produces good results on high-resolution devices like printers, and is the best mode to use for printing. However, when character widths are rounded for the screen, the results are generally poor. Characters are not well-separated and can collide or overlap at small font sizes.

`B_STRING_SPACING`
Keeps the string at the same width as it would have for `B_CHAR_SPACING`, but optimizes the position of each character within that space. The position of a character depends on the surrounding characters and the overall width of the string. Collisions are unlikely in this mode, but because the width of the string constrains what can be done, characters may touch each other.

This mode is preferred when it's important to have the screen match the printed page—for example, to have lines break on-screen where they will break when the display is printed. As the user types new characters into a line of text, the application must redraw the entire line to add each character. The characters in the line may therefore appear to "jiggle" or jump around as new ones are added. New optimal positions are calculated for each character as the width and composition of the string changes.

`B_BITMAP_SPACING`
Calculates the width of each character according to its bitmap appearance on-screen. The widths are chosen for optimal spacing, so that characters never collide and rarely touch. This mode increases the `B_CHAR_SPACING` width of a string if necessary to keep characters separated. (For a small-sized bold font, it may increase the string width substantially.)

In this mode, the spacing between characters is regular and not contextually dependent. Character widths are integral values. This is the best mode for drawing small amounts of text in the user interface; it's the mode that BTextView objects use and it works for both proportional and fixed-width fonts. However, the spacing of text shown on-screen won't correspond to the spacing when the text is printed in `B_CHAR_SPACING` mode.

B_FIXED_SPACING
Positions characters according to a constant, integral width. This mode can only be used with fixed-width fonts (fonts with the **B_IS_FIXED** flag set); trying to use **B_FIXED_SPACING** on other fonts will result in **B_CHAR_SPACING** being used by default. All characters have the same escapement.

The **B_CHAR_SPACING** mode is the preferred mode for printing. It's also somewhat faster than **B_STRING_SPACING** or **B_BITMAP_SPACING**. In all modes other than **B_STRING_SPACING**, it's possible to change the character displayed at the end of a string by erasing it and drawing a new character. However, in **B_STRING_SPACING** mode, it's necessary to erase the entire string and redraw it. The longer the string, the better the results.

The **B_STRING_SPACING** and **B_BITMAP_SPACING** modes are relevant only for font sizes in a range of about 7.0 points to 18.0 points. Above that range, **B_CHAR_SPACING** achieves reasonable results on-screen and may be used even where one of the other two modes is specified. Below that range, the screen resolution isn't great enough for the different modes to produce significantly different results, so again **B_CHAR_SPACING** is used.

In addition, **B_CHAR_SPACING** is always used for rotated or sheared text and when antialiasing is disabled.

See also: **BView::DrawString()**, **GetEscapements()**

- Shear() see SetShear()
- Size() see SetSize()
- Spacing() see SetSpacing()

- StringWidth() , GetStringWidths()

float **StringWidth(**const char **string**) const float **StringWidth(**const char **string**, int32 *length*) const
void **GetStringWidths(**const char **stringArray**[], const int32 *lengthArray*[], int32 *numStrings*, float *widthArray*[]) const

StringWidth() returns how much room is required to draw a *string* in the font. It measures the characters encoded in *length* bytes of the string—or, if no *length* is specified, the entire string up to the null character, '0', which terminates it. The return value totals the width of all the characters in coordinate units; it's the length of the baseline required to draw the string.

GetStringWidth() provides the same information for a group of strings. It works its way through the *stringArray* looking at a total of *numStrings*. For each string, it gets the length at the corresponding index from the *lengthArray* and places the width of the string in the *widthArray* at the same index.

These functions take all the attributes of the font—including family, style, size, and spacing—into account.

See also: `BView::StringWidth()`

Operators

= (assignment)

```
BFont& operator =(const BFont&)
```

Assigns one BFont object to another. After the assignment, the two objects are identical to each other and do not share any data.

== (equality), != (inequality)

```
bool operator ==(const BFont&) const
bool operator !=(const BFont&) const
```

These operators test whether two BFont objects are identical in all respects. If all settable font attributes are the same in both objects, they're equal. If not, they're unequal.

Constants

font_metric_mode

Declared in: be/interface/Font.h

Constant	Description
`B_SCREEN_METRIC`	The screen metric.
`B_PRINTING_METRIC`	The printing metric.

The font_metric_mode constants above indicate whether a font calculation should be done with the screen or the printer in mind.

font_file_format

Declared in: be/interface/Font.h

Constant	Description
`B_TRUETYPE_WINDOWS`	Microsoft Windows format TrueType font.
`B_POSTSCRIPT_TYPE1_WINDOWS`	Microsoft Windows format PostScript Type 1 font.

The font_file_format constants are used to specify what type of file a font was loaded from. Currently, only Microsoft Windows-format TrueType and PostScript Type 1 fonts are supported.

Defined Types

unicode_block

<be/interface/Font.h>

```
class unicode_block {
public:
   inline unicode_block();
   inline unicode_block(uint64 block2, uint64 block1);

   inline bool Includes(const unicode_block &block) const;
   inline unicode_block operator&(const unicode_block &block) const;
   inline unicode_block operator|(const unicode_block &block) const;
   inline unicode_block &operator=(const unicode_block &block);
   inline unicode_block operator==(const unicode_block &block) const;
   inline unicode_block operator!=(const unicode_block &block) const;

private:
   fData[2];
};
```

The unicode_block class describes the ranges of Unicode characters a font supports. You can get a unicode_block object for a font by calling the **Blocks()** function. Once you have this, you can check to see if a particular block is supported, or compare it to another block to see if it's inclusive, equal, unequal, and so forth.

In general, you won't instantiate a unicode_block object on your own.

The `unicode_block::Includes()` function lets you determine if another block is a subset of the unicode_block object.

There are a number of predefined Unicode blocks, as follows:

- **B_BASIC_LATIN_BLOCK**
- **B_LATIN1_SUPPLEMENT_BLOCK**
- **B_LATIN_EXTENDED_A_BLOCK**
- **B_LATIN_EXTENDED_B_BLOCK**
- **B_IPA_EXTENSIONS_BLOCK**
- **B_SPACING_MODIFIER_LETTERS_BLOCK**
- **B_COMBINING_DIACRITICAL_MARKS_BLOCK**

- B_BASIC_GREEK_BLOCK
- B_GREEK_SYMBOLS_AND_COPTIC_BLOCK
- B_CYRILLIC_BLOCK
- B_ARMENIAN_BLOCK
- B_BASIC_HEBREW_BLOCK
- B_HEBREW_EXTENDED_BLOCK
- B_BASIC_ARABIC_BLOCK
- B_ARABIC_EXTENDED_BLOCK
- B_DEVANAGARI_BLOCK
- B_BENGALI_BLOCK
- B_GURMUKHI_BLOCK
- B_GUJARATI_BLOCK
- B_ORIYA_BLOCK
- B_TAMIL_BLOCK
- B_TELUGU_BLOCK
- B_KANNADA_BLOCK
- B_MALAYALAM_BLOCK
- B_THAI_BLOCK
- B_LAO_BLOCK
- B_BASIC_GEORGIAN_BLOCK
- B_GEORGIAN_EXTENDED_BLOCK
- B_HANGUL_JAMO_BLOCK
- B_LATIN_EXTENDED_ADDITIONAL_BLOCK
- B_GREEK_EXTENDED_BLOCK
- B_GENERAL_PUNCTUATION_BLOCK
- B_SUPERSCRIPTS_AND_SUBSCRIPTS_BLOCK
- B_CURRENCY_SYMBOLS_BLOCK
- B_COMBINING_MARKS_FOR_SYMBOLS_BLOCK

- B_LETTERLIKE_SYMBOLS_BLOCK
- B_NUMBER_FORMS_BLOCK
- B_ARROWS_BLOCK
- B_MATHEMATICAL_OPERATORS_BLOCK
- B_MISCELLANEOUS_TECHNICAL_BLOCK
- B_CONTROL_PICTURES_BLOCK
- B_OPTICAL_CHARACTER_RECOGNITION_BLOCK
- B_ENCLOSED_ALPHANUMERICS_BLOCK
- B_BOX_DRAWING_BLOCK
- B_BLOCK_ELEMENTS_BLOCK
- B_GEOMETRIC_SHAPES_BLOCK
- B_MISCELLANEOUS_SYMBOLS_BLOCK
- B_DINGBATS_BLOCK
- B_CJK_SYMBOLS_AND_PUNCTUATION_BLOCK
- B_HIRAGANA_BLOCK
- B_KATAKANA_BLOCK
- B_BOPOMOFO_BLOCK
- B_HANGUL_COMPATIBILITY_JAMO_BLOCK
- B_CJK_MISCELLANEOUS_BLOCK
- B_ENCLOSED_CJK_LETTERS_AND_MONTHS_BLOCK
- B_CJK_COMPATIBILITY_BLOCK
- B_HANGUL_BLOCK
- B_HIGH_SURROGATES_BLOCK
- B_LOW_SURROGATES_BLOCK
- B_CJK_UNIFIED_IDEOGRAPHS_BLOCK
- B_PRIVATE_USE_AREA_BLOCK
- B_CJK_COMPATIBILITY_IDEOGRAPHS_BLOCK
- B_ALPHABETIC_PRESENTATION_FORMS_BLOCK

- **B_ARABIC_PRESENTATION_FORMS_A_BLOCK**
- **B_COMBINING_HALF_MARKS_BLOCK**
- **B_CJK_COMPATIBILITY_FORMS_BLOCK**
- **B_SMALL_FORM_VARIANTS_BLOCK**
- **B_ARABIC_PRESENTATION_FORMS_B_BLOCK**
- **B_HALFWIDTH_AND_FULLWIDTH_FORMS_BLOCK**
- **B_SPECIALS_BLOCK**
- **B_TIBETAN_BLOCK**

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFont Function Summary

Function Summary

Function	Description
!=	Compares two BFont objects for inequality
=	Assigns one BFont object to another
==	Compares two BFont objects for equality
Blocks()	Returns a unicode_block object indicating which Unicode blocks the font supports
BoundingBox()	Returns the bounding box of the entire font
CountTuned()	Returns number of tuned fonts for the family and style
Direction()	Returns direction in which the text is to be read
Encoding()	Returns character encoding scheme
Face()	Returns secondary font characteristics
FamilyAndStyle()	Returns internal representation of font family and style
FileFormat()	Returns the file format of

	the font
Flags()	Returns font behavior
BFont()	Constructor
GetBoundingBoxesAsGlyphs()	Returns bounding boxes for a group of characters' individual glyphs
GetBoundingBoxesAsString()	Returns bounding boxes for a group of characters as a string
GetBoundingBoxesForStrings()	Returns bounding boxes for a group of strings
GetEdges()	Returns edges of characters in a string
GetEscapements()	Returns escapements of characters in a string
GetFamilyAndStyle()	Returns family and style of font
GetGlyphShapes()	Returns BShape objects in the shape of specified glyphs
GetHasGlyphs()	Returns which characters in an array have glyphs in the font
GetHeight()	Returns font's height information
GetStringWidths()	Returns numbers of pixels required to draw strings
GetTruncatedStrings()	Truncates strings to fit within a horizontal bounds

GetTunedInfo()	Returns configuration of a tuned font
IsFixed()	Reports whether the font is fixed spacing or not
IsFullAndHalfFixed()	Unsupported in the current release
PrintToStream()	Writes font information to standard output
Rotation()	Returns baseline rotation
SetEncoding()	Sets character encoding scheme
SetFace()	Sets secondary font characteristics
SetFamilyAndFace()	Sets family and face of font
SetFamilyAndStyle()	Sets family and style of font
SetFlags()	Sets font behavior
SetRotation()	Sets baseline rotation
SetShear()	Sets shear
SetSize()	Sets font point size
SetSpacing()	Sets horizontal spacing mode
Shear()	Returns shear
Size()	Returns font point size
Spacing()	Returns horizontal spacing mode
StringWidth()	Returns number of pixels required to draw string

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView

Derived from: BHandler
Declared in: be/interface/View.h
Library: libbe.so
Summary: *more...*

The BView class is one of the largest in the BeOS class library. Because of this, the BView documentation is split across a number of files:

- BView Introduction is a brief intro to the BView class.
- BView General Functions describes those functions that you use no matter what you're doing with your BView object.
- BView Drawing-Related Functions: Anything related to drawing subdivided into Primitive Drawing Functions (to draw lines, circles, polygons, etc.) and Other Drawing Functions (such as picture recording, and bitmap functions).
- BView Graphics State Functions modify or report on the "graphics state," a set of variables (colors, pen position, scaling, etc.) that control how drawing takes place in the BView.
- BView Hook Functions gives a list of the BView functions that can be implemented in a subclass.

- BView View Hierarchy Functions relate to the nested tree of views associated with a window.
- BView Input-Related Functions monitor and process input activity.
- Scripting and Archival describes the scripting suites and properties, and archived fields supported by BView.

For information about a specific member function, click on the function name below:

- **BView()**, **~BView()**
- **AddChild()**, **AddLine()**, **AllAttached()**, **AllDetached()**, **AppendToPicture()**, **AttachedToWindow()**
- **BeginLineArray()**, **BeginPicture()**, **BeginRectTracking()**, **Bounds()**
- **ChildAt()**, **ClearViewBitmap()**, **ClearViewOverlay()**, **ClipToPicture()**, **ClipToInversePicture()**, **ConstrainClippingRegion()**, **ConvertFromParent()**, **ConvertFromScreen()**, **ConvertToParent()**, **ConvertToScreen()**, **CopyBits()**, **CountChildren()**
- **DetachedFromWindow()**, **DragMessage()**, **Draw()**, **DrawAfterChildren()**, **DrawBitmap()**, **DrawBitmapAsync()**, **DrawChar()**, **DrawingMode()**, **DrawPicture()**, **DrawPictureAsync()**, **DrawString()**
- **EndLineArray()**, **EndPicture()**, **EndRectTracking()**, **EventMask()**

- FindView(), FillArc(), FillBezier(), FillEllipse(), FillPolygon(), FillRect(), FillRegion(), FillRoundRect(), FillShape(), FillTriangle(), Flags(), Flush(), ForceFontAliasing(), Frame(), FrameMoved(), FrameResized()
- GetClippingRegion(), GetFont(), GetFontHeight(), GetMouse(), GetPreferredSize(), GetStringWidths(), GetSupportedSuites()
- Hide(), HighColor()
- Invalidate(), InvertRect(), IsFocus(), IsHidden(), IsPrinting()
- KeyDown(), KeyUp()
- LeftTop(), LineCapMode(), LineJoinMode(), LineMiterLimit(), LowColor()
- MakeFocus(), MessageReceived(), MouseDown(), MouseMoved(), MouseUp(), MoveBy(), MoveTo(), MovePenBy()
- MovePenTo()
- NextSibling()
- Origin()
- Parent(), PenLocation(), PenSize(), PopState(), PreviousSibling(), Pulse(), PushState()
- RemoveChild(), RemoveSelf(), ResizeBy(), ResizeTo(), ResizeToPreferred(), ResizingMode(), ResolveSpecifier()

- ScrollBar(), ScrollBy(), ScrollTo(), SetDrawingMode(), SetEventMask(), SetFlags(), SetFont(), SetFontSize(), HTMLSetHighColor(), HTMLSetLowColor(), SetViewColor(), SetLineMode(), SetMouseEventMask() SetOrigin(), SetPenSize(), SetResizingMode(), SetScale(), SetViewBitmap(), SetViewCursor(), SetViewOverlay(), Show(), StringWidth(), StrokeArc(), StrokeBezier(), StrokeEllipse(), StrokePolygon(), StrokeRect(), StrokeRegion(), StrokeRoundRect(), StrokeTriangle(), Sync()

- TargetedByScrollView(), TruncateString()

- ViewColor()

- Window(), WindowActivated()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Summary

The Interface Kit Table of Contents The Interface Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Introduction

Derived from: BHandler
Declared in: be/interface/View.h
Library: libbe.so
Summary: *more...*

A BView object represents a rectangular area within a window. The object draws within this rectangle and responds to user events (mouse clicks, key presses) that are directed at the window.

Most of the other classes in the Interface Kit inherit from BView. You can use these objects off-the-shelf, but for most applications, you'll also need to create some BView subclasses, and create instances of these subclasses.

Developing a BView Subclass

To define special behavior for a BView--such as the type of behavior that might be needed for the canvas area of a painting program--you define your own subclass of BView, and in that subclass implement the appropriate *hook functions* of the BView class. A hook function is a function that is called by the BeOS internals in response to certain events, and that is intended specifically for definition in a user-defined subclass. For example, the BView::MouseDown() hook function is called on a BView instance every time a mouse button is clicked while the mouse is inside the view. The default BView::MouseDown() function does nothing, but you can override this with your own implementation, to respond to mouse-down events as you wish.

The following are the most commonly used hook functions. Other hook functions provide for changes in the size or structure of views and windows, changes in the input focus, and so forth. For a complete list and reference of hook functions, see BView Hook Functions.

`Draw()`

Called whenever the contents of the view need to be drawn or redrawn.

`KeyDown()`

Called when a keyboard key is pressed.

`KeyUp()`

Called when a keyboard key is released.

`MouseDown()`

Called when a mouse button is clicked while the mouse cursor is in the view.

`MouseMoved()`

Called when the mouse cursor enters, exits, or moves within the view.

The View Structure

A newly created view is an "orphan"--it won't appear onscreen, and can't be used for much, because it isn't associated with an onscreen "parent". To rectify this situation, and give your new view a warm and loving home, invoke the AddChild() method of an existing BWindow or BView object to add the new view as a child, i.e.

```
existingWindowOrView.AddChild(yourNewView);
```

Ultimately, all views displayed in a window end up as a tree of views, with the window as the root of the tree.

See the BView View Hierarchy Functions section for BView functions related to this view structuring.

Locking the Window

Most BView functions expect the view's BWindow to be locked. To find a view's BWindow and lock/unlock it, you first call BView::Window(), and then call the **LockLooper()** and **UnlockLooper()** functions (defined by BHandler, inherited by BWindow):

```
if (window. LockLooper() ) {
   . . .
   window.UnlockLooper();
}
```

BView functions check to ensure sure the caller has the window locked. If the window isn't properly locked, they print warning messages and fail.

Whenever the system calls a BView hook function, it automatically locks the BWindow for you —**you never have to lock the BWindow in the implementation of BView hook function**.

Focus

To facilitate keyboard navigation of views, BView provides integral support for the concept of *focus*. The view that has the focus is the one whose `KeyDown()` function is called to process keyboard events. Only one view in a window can have the focus at any given time

From the user's point-of-view, the **tab** key rotates the focus from one view to the next through the navigation group, cycling back to the first view if the **tab** key is pressed while the last targetable view in the group has the focus. **shift**+**tab** cycles the focus backward. Holding down the control key while cycling groups causes the focus to jump from one navigation group to the next.

When a view has the focus, some sort of indicator should be drawn to inform the user that the view is the focus. Typically, this involves drawing a line under a label in the view, or possibly drawing a box around the view or some portion of it. The global `keyboard_navigation_color()` function should be used to obtain the color used to draw the focus indicator.

The view's `MakeFocus()` function is called to specify whether or not the control has the focus; it's called with an argument of `true` if the control has the focus, and `false` if it's not the focus; `MakeFocus()` calls the previously-focused view's `MakeFocus()` function to inform that view that it's not the focus anymore. You can augment `MakeFocus()` in a subclass if you need to take notice when the view becomes the focus (or loses the focus). For example, you may need to draw or erase the keyboard navigation indicator.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Hook Functions

Derived from: BHandler
Declared in: be/interface/View.h
Library: libbe.so
Summary: *more...*

The functions in this section are called in response to particular events, and are intended to be overridden in a BView subclass. (The default implementations of the BView hook functions do nothing.) They include functions to respond to input, draw when necessary, perform actions periodically, and various others.

Hook Functions

- AllAttached() see AttachedToWindow()
- AllDetached() see DetachedFromWindow()

- AttachedToWindow() , AllAttached()

virtual void **AttachedToWindow**(void)
virtual void **AllAttached**(void)

Implemented by derived classes to complete the initialization of the BView when it's assigned to a window. A BView is assigned to a window when it, or one of its ancestors in the view hierarchy, becomes a child of a view already attached to a window.

AttachedToWindow() is called immediately after the BView is formally made a part of the window's view hierarchy and after it has become known to the Application Server and its graphics parameters are set. The **Window()** function can identify which BWindow the BView belongs to.

All of the BView's children, if it has any, also become attached to the window and receive their own **AttachedToWindow()** notifications. Parents receive the notification before their children, but only after all views have become attached to the window and recognized as part of the window's view hierarchy. This function can therefore depend on all ancestor and descendant views being in place.

For example, **AttachedToWindow()** can be implemented to set a view's background color to the same color as its parent, something that can't be done before the view belongs to a window and knows who its parent is.

```
void MyView::AttachedToWindow()
{
   if ( Parent() )
      SetViewColor(Parent()->ViewColor());
   baseClass::AttachedToWindow();
}
```

The **AllAttached()** notification follows on the heels of **AttachedToWindow()**, but works its way up the view hierarchy rather than down. When **AllAttached()** is called for a BView, all its descendants have received both **AttachedToWindow()** and **AllAttached()** notifications. Therefore, parent views can depend on any calculations that their children make in either function. For example, a parent can resize itself to fit the size of its children, where their sizes depend on calculations done in **AttachedToWindow()**.

The default (BView) version of both these functions are empty.

See also: `AddChild()`, `Window()`

DetachedFromWindow() , AllDetached()

virtual void **DetachedFromWindow**(void)
virtual void **AllDetached**(void)

Implemented by derived classes to make any adjustments necessary when the BView is about to be removed from a window's view hierarchy. These two functions parallel the more commonly implemented `AttachedToWindow()` and `AllAttached()` functions.

`DetachedFromWindow()` notifications work their way down the hierarchy of views being detached, followed by `AllDetached()` notifications, which work their way up the hierarchy. The second function call permits an ancestor view to take actions that depend on calculations a descendant might have to make when it's first notified of being detached.

The BView is still attached to the window when both functions are called.

See also: `AttachedToWindow()`

Draw()

virtual void **Draw**(BRect *updateRect*)

Implemented by derived classes to draw the *updateRect* portion of the view. The update rectangle is stated in the BView's coordinate system.

`Draw()` is called as the result of update messages whenever the view needs to present itself on-screen. This may happen when:

- The window the view is in is first shown on-screen, or shown after being hidden (see the BWindow version of the `Hide()` function).
- The view is made visible after being hidden (see BView's `Hide()` function).
- Obscured parts of the view are revealed, as when a window is moved from in front of the view or an image is dragged across it.
- The view is resized.
- The contents of the view are scrolled (see `ScrollBy()`).
- A child view is added, removed, or resized.
- A rectangle has been invalidated that includes at least some of the view (see `Invalidate()`).
- `CopyBits()` can't completely fill a destination rectangle within the view.

Draw() is also called from a BPrintJob object's DrawView() function to draw the view on a printed page. IsPrinting() returns `true` when the BView is drawing for the printer and `false` when it's drawing to the screen. When printing, you may want to recalculate layouts, substitute fonts, turn antialiasing off, scale the size of a coordinate unit, or make other adjustments to ensure the quality of the printed image.

When drawing to the screen, the *updateRect* is the smallest rectangle that encloses the current clipping region for the view. Since the Application Server won't render anything on-screen that's outside the clipping region, an application will be more efficient if it avoids producing drawing instructions for images that don't intersect with the rectangle. For still more efficiency and precision, you can ask for the clipping region itself (by calling GetClippingRegion()) and confine drawing to images that intersect with it.

When printing, the *updateRect* matches the rectangle passed to DrawView() and may lie outside the clipping region. The clipping region is not enforced for printing, but the Print Server clips the BView's drawing to the specified rectangle.

See also: BWindow::UpdateIfNeeded(), Invalidate(), GetClippingRegion(), BPrintJob::DrawView(), IsPrinting()

DrawAfterChildren()

virtual void **Draw(**BRect *updateRect***)**

This function is similar (in fact, almost identical) to Draw(). The only difference is that DrawAfterChildren() is called *after* all children have drawn during a screen update. This is in contrast to Draw(), which draws before any children have drawn. In general, Draw() will be used for almost all of your drawing needs; DrawAfterChildren() is intended for use in the rare circumstances where you wish a view to be able to draw on top of its child views.

Other details are as for Draw().

See also: Draw()

FrameMoved()

virtual void **FrameMoved(**BPoint *newLocation***)**

Implemented by derived classes to respond to a notification that the view has moved within its parent's coordinate system. *newLocation* gives the new location, within the coordinate system of the view's window, of the left top corner of the BView's frame rectangle.

FrameMoved() is called only if the B_FRAME_EVENTS flag is set and the BView is attached to a window.

If the view is both moved and resized, FrameMoved() is called before FrameResized(). This might happen, for example, if the BView's automatic resizing mode is a combination of

B_FOLLOW_TOP_BOTTOM and B_FOLLOW_RIGHT and its parent is resized both horizontally and vertically.

The default (BView) version of this function is empty.

Currently, FrameMoved() is also called when a hidden window is shown on-screen.

See also: MoveBy(), BWindow::FrameMoved(), SetFlags()

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Implemented by derived classes to respond to a notification that the view has been resized. The arguments state the new *width* and *height* of the view. The resizing could have been the result of a user action (resizing the window) or of a programmatic one (calling ResizeTo() or ResizeBy()).

FrameResized() is called only if the B_FRAME_EVENTS flag is set and the BView is attached to a window.

BView's version of this function is empty.

See also: ResizeBy(), BWindow::FrameResized(), SetFlags()

GetPreferredSize() , ResizeToPreferred()

virtual void **GetPreferredSize(**float **width*, float **height***)**

virtual void **ResizeToPreferred(**void**)**

GetPreferredSize() is implemented by derived classes to write the preferred width and height of the view into the variables the *width* and *height* arguments refer to. Derived classes generally make this calculation based on the view's contents. For example, a BButton object reports the optimal size for displaying the button border and label given the current font.

ResizeToPreferred() resizes the BView's frame rectangle to the preferred size, keeping its left and top sides constant.

See also: ResizeTo()

KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes***)**

Implemented by derived classes to respond to a B_KEY_DOWN message reporting keyboard input. Whenever a BView is the focus view of the active window, it receives a KeyDown() notification for each character the user types, except for those that:

- Are produced while a Command key is held down. Command key events are interpreted as keyboard shortcuts.
- Are produced by the Tab key when an Option key is held down. Option-Tab events are invariably interpreted as instructions to change the focus view (for keyboard navigation); they work even where Tab alone does not.
- Can operate the default button in a window. The BButton object's KeyDown() function is called, rather than the focus view's.

The first argument, *bytes*, is an array that encodes the character mapped to the key the user pressed. The second argument, *numBytes*, tells how many bytes are in the array; there will always be at least one. The *bytes* value follows the character encoding of the BView's font. Typically, the encoding is Unicode UTF-8 (B_UNICODE_UTF8), so there may be more than one byte per character. The *bytes* array is not null-terminated; '0' is a valid character value.

The character value takes into account any modifier keys that were held down or keyboard locks that were on at the time of the keystroke. For example, Shift-*i* is reported as uppercase 'I' (0x49) and Control-*i* is reported as a B_TAB (0x09).

Single-byte characters can be tested against ASCII codes and these constants:

B_BACKSPACE	B_LEFT_ARROW	B_INSERT
B_ENTER	B_RIGHT_ARROW	B_DELETE
B_RETURN	B_UP_ARROW	B_HOME
B_SPACE	B_DOWN_ARROW	B_END
B_TAB		B_PAGE_UP
B_ESCAPE	B_FUNCTION_KEY	B_PAGE_DOWN

B_ENTER and B_RETURN are the same character, a newline ('n').

Only keys that generate characters produce key-down events; the modifier keys on their own do not.

You can determine which modifier keys were being held down at the time of the event by calling BLooper's CurrentMessage() function and looking up the "modifiers" entry in the BMessage it returns. If the *bytes* character is B_FUNCTION_KEY and you want to know which key produced the character, you can look up the "key" entry in the BMessage and test it against these constants:

B_F1_KEY	B_F6_KEY	B_F11_KEY
B_F2_KEY	B_F7_KEY	B_F12_KEY
B_F3_KEY	B_F8_KEY	B_PRINT_KEY (Print Screen)
B_F4_KEY	B_F9_KEY	B_SCROLL_KEY (Scroll Lock)

B_F5_KEY	B_F10_KEY	B_PAUSE_KEY

For example:

```
if ( bytes[0] == B_FUNCTION_KEY ) {
   BMessage *msg = Window()->CurrentMessage();
   if ( msg ) {
      int32 key;
      msg->FindInt32("key", &key);
      switch ( key ) {
      case B_F1_KEY:
         . . .
         break;
      case B_F2_KEY:
         . . .
         break;
      . . .
      }
   }
}
```

The BView version of KeyDown() handles keyboard navigation from view to view through B_TAB characters. If the view you define is navigable, its KeyDown() function should permit B_SPACE characters to operate the object and perhaps allow the arrow keys to navigate inside the view. It should also call the inherited version of KeyDown() to enable between-view navigation. For example:

```
void MyView::KeyDown(const char *bytes, int32 numBytes)
{
   if ( numBytes == 1 ) {
      switch ( bytes[0] ) {
      case B_SPACE:
         /* mimic a click in the view*/
         break;
      case B_RIGHT_ARROW:
         /* move one position to the right in the view*/
         break;
      case B_LEFT_ARROW:
         /* move one position to the left in the view*/
         break;
      default:
         baseClass::KeyDown(bytes, numBytes);
         break;
      }
   }
}
```

If your BView is navigable but needs to respond to **B_TAB** characters—for example, if it permits users to insert tabs in a text string—its **KeyDown()** function should simply grab the characters and not pass them to the inherited function. Users will have to rely on the Option-Tab combination to navigate from your view.

See also: the **Keyboard Information** appendix, "B_KEY_DOWN" in the **Message Protocols** appendix, **BWindow::SetDefaultButton()**, **modifiers()**

KeyUp()

virtual void **KeyUp(**const char ***bytes*, int32 *numBytes***)**

Implemented by derived classes to respond to a **B_KEY_UP** message reporting that the user released a key on the keyboard. The same set of keys that produce **B_KEY_DOWN** messages when they're pressed produce **B_KEY_UP** messages when they're released. The *bytes* and *numBytes* arguments encode the character mapped to the key the user released; they work exactly like the same arguments passed to **KeyDown()**.

Some **B_KEY_DOWN** messages are swallowed by the system and are never dispatched by calling **KeyDown()**; others are dispatched, but not to the focus view. In contrast, all **B_KEY_UP** messages are dispatched by calling **KeyUp()** for the focus view of the active window. Since the focus view and active window can change between the time a key is pressed and the time it's released, this may or may not be the same BView that was notified of the **B_KEY_DOWN** message.

See also: **KeyDown()**, "B_KEY_DOWN" in the **Message Protocols** appendix

---

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

MouseDown() is a hook function that's invoked when the user depresses a mouse button (or other pointing device button, not including joysticks). The location of the cursor at the time of the event is given by *point* in the BView's coordinates. See **B_MOUSE_DOWN** for the message format. Also see **SetMouseEventMask()** for information on extending the view's event mask while the mouse is being held down.

The BView version of **MouseDown()** is empty.

MouseMoved()

virtual void **MouseMoved(**BPoint *point*, uint32 *transit*, const BMessage **message***)**

Implemented by derived classes to respond to reports of mouse-moved events associated with the view. As the user moves the cursor over a window, the Application Server generates a continuous stream of messages reporting where the cursor is located.

The first argument, *point*, gives the cursor's new location in the BView's coordinate system. The second argument, *transit*, is one of four constants,

B_ENTERED_VIEW
B_INSIDE_VIEW
B_EXITED_VIEW
B_OUTSIDE_VIEW

which explains whether the cursor has just entered the visible region of the view, is now inside the visible region having previously entered, has just exited from the view, or is currently outside the visible region of the view. When the cursor passes from one view to another, **MouseMoved()** is called on each of the BViews, once with a *transit* code of **B_EXITED_VIEW** and the other with a code of **B_ENTERED_VIEW**.

If the user is dragging a bundle of information from one location to another, the final argument, *message*, is a pointer to the BMessage object that holds the information. If a message isn't being dragged, *message* is NULL.

The default version of **MouseMoved()** is empty.

MouseUp()

virtual void **MouseUp(**BPoint *point***)**

Implemented by derived classes to respond to a message reporting a mouse-up event within the view. The location of the cursor at the time of the event is given by *point* in the BView's coordinates.

Pulse()

virtual void **Pulse(**void**)**

Implemented by derived classes to do something at regular intervals. Pulses are regularly timed events, like the tick of a clock or the beat of a steady pulse. A BView receives **Pulse()** notifications when no other messages are pending, but only if it asks for them with the **B_PULSE_NEEDED** flag.

The interval between **Pulse()** calls can be set with BWindow's **SetPulseRate()** function. The default interval is around 500 milliseconds. The pulse rate is the same for all views within a window, but can vary between windows.

Derived classes can implement a **Pulse()** function to do something that must be repeated continuously. However, for time-critical actions, you should implement your own timing mechanism.

The BView version of this function is empty.

See also: `SetFlags()`, the BView constructor, `BWindow::SetPulseRate()`

ResizeToPreferred() see GetPreferredSize()

TargetedByScrollView()

virtual void **TargetedByScrollView(**BScrollView ***scroller*****)**

Implemented by derived classes to respond to a notification that the BView has become the target of the *scroller* BScrollView object. This function is called when the BScrollView sets its target, which it does on construction. The target is the object whose contents will be scrolled.

BView's implementation of this function is empty.

See also: the BScrollView class, and the various scrolling-related functions in BView Input-Related Functions.

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Implemented by derived classes to take whatever steps are necessary when the BView's window becomes the active window, or when the window gives up that status. If *active* is `true`, the window has become active. If *active* is `false`, it no longer is the active window.

All objects in the view hierarchy receive `WindowActivated()` notifications when the status of the window changes.

BView's version of this function is empty.

See also: `BWindow::WindowActivated()`

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Hook Functions Function Summary

Function Summary

Function	Description
AllAttached()	Hook called when all the views have been attached to a window
AllDetached()	Hook called after all views have been detached from the window
AttachedToWindow()	Hook called when the view is attached to a window
DetachedFromWindow()	Hook called when the view is detached from the window
DrawAfterChildren()	Hook to update the view after child views have drawn; allows drawing on top of children.
Draw()	Hook to update the view
FrameMoved()	Hook called when the view moves within its parent's coordinate system
FrameResized()	Hook called when the view frame is resized
GetPreferredSize()	Hook returning the preferred size of the view
KeyDown()	Hook called in response to a **`B_KEY_DOWN`** message
KeyUp()	Hook called in response to a **`B_KEY_UP`** message
MouseDown()	Hook called in response to a

	B_MOUSE_DOWN message
MouseMoved()	Hook called in response to a **B_MOUSE_MOVED** message
MouseUp()	Hook called in response to a **B_MOUSE_UP** message
Pulse()	Hook called at regular intervals
ResizeToPreferred()	Resizes frame to the preferred size
TargetedByScrollView()	Hook called when the view becomes targeted by a BScrollView
WindowActivated()	Hook called when the parent window is activated or deactivated

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

General BView Functions

This section describes BView functionality which is of general use, or which does not fit neatly into any of the other BView document sections describing the BView functions.

Constructor and Destructor

BView()

BView(BRect *frame*, const char **name*, uint32 *resizingMode*, uint32 *flags*)
BView(BMessage **archive*)

Sets up a view with the *frame* rectangle, which is specified in the coordinate system of its eventual parent, and assigns the BView an identifying *name*, which can be NULL.

When it's created, a BView doesn't belong to a window and has no parent. It's assigned a parent by having another BView adopt it with the **AddChild()** function. If the other view is in a window, the BView becomes part of that window's view hierarchy. A BView can be made a child of the window's top view by calling BWindow's version of the **AddChild()** function.

When the BView gains a parent, the values in *frame* are interpreted in the parent's coordinate system. The sides of the view must be aligned on screen pixels. Therefore, the *frame* rectangle should not contain coordinates with fractional values. Fractional coordinates will be rounded to the first lower whole number (for example 1.2 will be rounded down to 1.0).

The *resizingMode* mask determines the behavior of the view when its parent is resized. It should combine one constant for horizontal resizing,

B_FOLLOW_LEFT
B_FOLLOW_RIGHT
B_FOLLOW_LEFT_RI GHT
B_FOLLOW_H_CENT ER

with one for vertical resizing:

B_FOLLOW_TOP
B_FOLLOW_BOTTOM
B_FOLLOW_TOP_BOTT OM
B_FOLLOW_V_CENTER

For example, if **B_FOLLOW_LEFT** is chosen, the margin between the left side of the view and the left side of its parent will remain constant—the view will "follow" the parent's left side. Similarly, if **B_FOLLOW_RIGHT** is chosen, the view will follow the parent's right side. If **B_FOLLOW_H_CENTER** is chosen, the view will maintain a constant relationship to the horizontal center of the parent.

If the constants name opposite sides of the view rectangle—left and right, or top and bottom—the view will necessarily be resized in that dimension when the parent is. For example, **B_FOLLOW_LEFT_RIGHT** means that the margin between the left side of the view and left side of the parent will remain constant, as will the margin between the right side of the view and the

right side of the parent. As the parent is resized horizontally, the child will be resized with it. Note that **B_FOLLOW_LEFT_RIGHT** is not the same as combining **B_FOLLOW_LEFT** and **B_FOLLOW_RIGHT** , an illegal move. The *resizingMode* mask can contain only one horizontal constant and one vertical constant.

If a side is not mentioned in the mask, the distance between that side of the view and the corresponding side of the parent is free to fluctuate. This may mean that the view will move within its parent's coordinate system when the parent is resized. **B_FOLLOW_RIGHT** plus **B_FOLLOW_BOTTOM**, for example, would keep a view from being resized, but the view will move to follow the right bottom corner of its parent whenever the parent is resized. **B_FOLLOW_LEFT** plus **B_FOLLOW_TOP** prevents a view from being resized *and* from being moved.

In addition to the constants listed above, there are two other possibilities:

B_FOLLOW_ALL_SI DES
B_FOLLOW_NONE

B_FOLLOW_ALL_SIDES is a shorthand for **B_FOLLOW_LEFT_RIGHT** and **B_FOLLOW_TOP_BOTTOM**. It means that the view will be resized in tandem with its parent, both horizontally and vertically.

B_FOLLOW_NONE behaves just like **B_FOLLOW_LEFT**|**B_FOLLOW_TOP**; the view maintains the same position in its parent's coordinate system, but not in the screen coordinate system.

Typically, a parent view is resized because the user resizes the window it's in. When the window is resized, the top view is too. Depending on how the *resizingMode* flag is set for the top view's children and for the descendants of its children, automatic resizing can cascade down the view hierarchy. A view can also be resized programmatically by the **ResizeTo()** and **ResizeBy()** functions.

The resizing mode can be changed after construction with the **SetResizingMode()** function.

The *flags* mask determines what kinds of notifications the BView will receive. It can be any combination of the following constants:

B_WILL_DRAW
Indicates that the BView does some drawing of its own and therefore can't be ignored when the window is updated. If this flag isn't set, the BView won't receive update notifications—its **Draw()** function won't be called—and it won't be erased to its background view color if the color is other than white.

B_PULSE_NEEDED
Indicates that the BView should receive **Pulse()** notifications.

B_FRAME_EVENTS
Indicates that the BView should receive **FrameResized()** and **FrameMoved()** notifications when its frame rectangle changes—typically as a result of the automatic resizing behavior described above. **FrameResized()** is called when the dimensions of the view change; **FrameMoved()** is called when the position of its left top corner in its parent's coordinate system changes.

B_FULL_UPDATE_ON_RESIZE
Indicates that the entire view should be updated when it's resized. If this flag isn't set, only the portions that resizing adds to the view will be included in the clipping region. This doesn't affect the view's children; their own flags determine when updates will occur.

B_NAVIGABLE
Indicates that the BView can become the focus view for keyboard actions. This flag makes it possible for the user to navigate to the view and put it in focus by pressing the Tab key. See "Keyboard Navigation" at the beginning of this chapter.

B_NAVIGABLE_JUMP
Marks the position of a group of views for keyboard navigation. By pressing Control-Tab, the user can jump from group to group. The focus lands on the first BView in the group that has the **B_NAVIGABLE** flag set. This may be the same BView that has the **B_NAVIGABLE_JUMP** marker, or the **B_NAVIGABLE_JUMP** BView may be the parent of a group of **B_NAVIGABLE** views.

B_SUBPIXEL_PRECISE
Instructs the rendering methods to use subpixel precision when drawing. If you don't set this flag, coordinates are rounded to the nearest unit.

If none of these constants apply, *flags* can be `NULL`. The flags can be reset after construction with the **SetFlags()** function.

See also: **SetResizingMode()**, **SetFlags()**, **BHandler::SetName()**

~BView()

virtual **~BView()**

Frees all memory the BView allocated, and ensures that each of the BView's descendants in the view hierarchy is also destroyed.

It's an error to delete a BView while it remains attached to a window. Call **RemoveChild()** or **RemoveSelf()** before using the `delete` operator.

See also: **RemoveChild()**

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*****)**

Returns a new BView object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the message doesn't contain archived data for a BView, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**

Member Functions

Bounds()

```
BRect Bounds(void) const
```

Returns the BView's bounds rectangle.

ConvertToParent() , ConvertFromParent()

```
BPoint ConvertToParent(BPoint localPoint) const
void ConvertToParent(BPoint *localPoint) const
BRect ConvertToParent(BRect localRect) const
void ConvertToParent(BRect *localRect) const
```

```
BPoint ConvertFromParent(BPoint parentPoint) const
void ConvertFromParent(BPoint *parentPoint) const
BRect ConvertFromParent(BRect parentRect) const
void ConvertFromParent(BRect *parentRect) const
```

These functions convert points and rectangles to and from the coordinate system of the BView's parent. `ConvertToParent()` converts *localPoint* or *localRect* from the BView's coordinate system to the coordinate system of its parent BView. `ConvertFromParent()` does the opposite; it converts *parentPoint* or *parentRect* from the coordinate system of the BView's parent to the BView's own coordinate system.

If the point or rectangle is passed by value, the function returns the converted value. If a pointer is passed, the conversion is done in place.

Both functions fail if the BView isn't attached to a window.

See also: `ConvertToScreen()`

ConvertToScreen() , ConvertFromScreen()

```
BPoint ConvertToScreen(BPoint localPoint) const
void ConvertToScreen(BPoint *localPoint) const
BRect ConvertToScreen(BRect localRect) const
void ConvertToScreen(BRect *localRect) const
```

```
BPoint ConvertFromScreen(BPoint screenPoint) const
void ConvertFromScreen(BPoint *screenPoint) const
BRect ConvertFromScreen(BRect screenRect) const
void ConvertFromScreen(BRect *screenRect) const
```

ConvertToScreen() converts *localPoint* or *localRect* from the BView's coordinate system to the global screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts *screenPoint* or *screenRect* from the screen coordinate system to the BView's local coordinate system.

If the point or rectangle is passed by value, the function returns the converted value. If a pointer is passed, the conversion is done in place.

The screen coordinate system has its origin, (0.0, 0.0), at the left top corner of the main screen.

Neither function will work if the BView isn't attached to a window.

See also: **BWindow::ConvertToScreen()**, **ConvertToParent()**

Flags() see SetFlags()

Frame()

BRect **Frame**(void) const

Returns the BView's frame rectangle. The frame rectangle is first set by the BView constructor and is altered only when the view is moved or resized. It's stated in the coordinate system of the BView's parent.

Hide() , Show()

virtual void **Hide**(void)
virtual void **Show**(void)

These functions hide a view and show it again.

Hide() makes the view invisible without removing it from the view hierarchy. The visible region of the view will be empty and the BView won't receive update messages. If the BView has children, they also are hidden.

Show() unhides a view that had been hidden. This function doesn't guarantee that the view will be visible to the user; it merely undoes the effects of **Hide()**. If the view didn't have any visible area before being hidden, it won't have any after being shown again (given the same conditions).

Calls to **Hide()** and **Show()** can be nested. For a hidden view to become visible again, the number of **Hide()** calls must be matched by an equal number of **Show()** calls.

However, **Show()** can only undo a previous **Hide()** call on the same view. If the view became hidden when **Hide()** was called to hide the window it's in or to hide one of its ancestors in the

view hierarchy, calling **Show()** on the view will have no effect. For a view to come out of hiding, its window and all its ancestor views must be unhidden.

Hide() and **Show()** can affect a view before it's attached to a window. The view will reflect its proper state (hidden or not) when it becomes attached. Views are created in an unhidden state.

See also: **BWindow::Hide()**, **IsHidden()**

IsFocus()

bool **IsFocus**(void) const

Returns `true` if the BView is the current focus view for its window, and `false` if it's not. The focus view changes as the user chooses one view to work in and then another—for example, as the user moves from one text field to another when filling out an on-screen form. The change is made programmatically through the **MakeFocus()** function.

See also: **BWindow::CurrentFocus()**, **MakeFocus()**

IsHidden()

bool **IsHidden**(void) const

Returns `true` if the view has been hidden by the **Hide()** function, and `false` otherwise.

This function returns `true` whether **Hide()** was called to hide the BView itself, to hide an ancestor view, or to hide the BView's window. When a window is hidden, all its views are hidden with it. When a BView is hidden, all its descendants are hidden with it.

If the view has no visible region—perhaps because it lies outside its parent's frame rectangle or is obscured by a window in front—this function may nevertheless return `false`. It reports only whether the **Hide()** function has been called to hide the view, hide one of the view's ancestors in the view hierarchy, or hide the window where the view is located.

If the BView isn't attached to a window, **IsHidden()** returns the state that it will assume when it becomes attached. By default, views are not hidden.

See also: **Hide()**

IsPrinting()

bool **IsPrinting**(void) const

Returns `true` if the BView is being asked to draw for the printer, and `false` if the drawing it produces will be rendered on-screen (or if the BView isn't being asked to draw at all).

This function's result is only reliable when called from within `Draw()` or `DrawChildren()` to determine whether the drawing it does is destined for the printer or the screen. When drawing to the printer, the BView may choose different parameters—such as fonts, bitmap images, or colors—than when drawing to the screen.

You should avoid calling this function from outside `Draw()` and `DrawChildren()`; however, if you absolutely have to do it, lock the view first. Failure to do so may bring up the debugger—if not in BeOS 5, it may in future versions of BeOS.

See also: the BPrintJob class, `Draw()` , `DrawChildren()`

LeftTop()

BPoint **LeftTop(**void**)** const

Returns the coordinates of the left top corner of the view—the smallest *x* and *y* coordinate values within the bounds rectangle.

See also: `BRect::LeftTop()`, `Bounds()`

Origin() see SetOrigin()

ResizeBy() , ResizeTo()

void **ResizeBy(**float *horizontal*, float *vertical***)**
void **ResizeTo(**float *width*, float *height***)**

These functions resize the view, without moving its left and top sides. `ResizeBy()` adds *horizontal* coordinate units to the width of the view and *vertical* units to the height. `ResizeTo()` makes the view *width* units wide and *height* units high. Both functions adjust the right and bottom components of the frame rectangle accordingly.

Since a BView's frame rectangle must be aligned on screen pixels, only integral values should be passed to these functions. Values with fractional components will be rounded to the nearest whole integer.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new size. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame and bounds rectangles.

If the view isn't attached to a window, its frame and bounds rectangles are adjusted, but its children, if any, don't get corresponding adjustments.

See also: `FrameResized()`, `MoveBy()`, `BRect::Width()`, `Frame()`

ResizingMode() see SetResizingMode()

SetFlags() , Flags()

virtual void **SetFlags(**uint32 *mask***)**
uint32 **Flags(**void**)** const

These functions set and return the flags that inform the Application Server about the kinds of notifications the BView should receive. The *mask* set by **SetFlags()** and the return value of **Flags()** is formed from combinations of the following constants:

B_WILL_DRAW
B_FULL_UPDATE_ON_RESIZE
B_FRAME_EVENTS
B_PULSE_NEEDED
B_NAVIGABLE
B_NAVIGABLE_JUMP
B_SUBPIXEL_PRECISE

The flags are first set when the BView is constructed; they're explained in the description of the BView constructor. The mask can be 0.

To set just one of the flags, combine it with the current setting:

```
myView->SetFlags(Flags() | B_FRAME_EVENTS);
```

See also: the BView constructor, **SetResizingMode()**

SetOrigin() , Origin()

void **SetOrigin(**BPoint *pt***)** void **SetOrigin(**float *x*, float *y***)**
BPoint **Origin(**void**)** const

Sets and retrieves the local origin of the BView's coordinate system.

The actual origin used by the Application Server is the sum of the local origin (as set by this method) and the origins stored on the state stack (properly scaled).

SetResizingMode() , ResizingMode()

virtual void **SetResizingMode(**uint32 *mode***)**

uint32 **ResizingMode(**void**)** const

These functions set and return the BView's automatic resizing mode. The resizing mode is first set when the BView is constructed. The various possible modes are explained where the constructor is described.

See also: the BView constructor, **SetFlags()**

SetViewCursor()

void **SetViewCursor(**const BCursor *_cursor_, bool sync=true**)** const

Sets the specified *cursor* as the view's cursor; while the mouse is inside the view, this cursor will be displayed (unless of course the cursor is hidden or obscured).

If *sync* is **true**, the Application Server will be synchronized by this call, forcing the change to take place immediately. If *sync* is **false**, the change will take place when the Application Server naturally gets to the change in its queue of pending requests.

Show() see Hide()

Sync() see Flush()

Window()

BWindow ***Window(**void**)** const

Returns the BWindow to which the BView belongs, or `NULL` if the BView isn't attached to a window. This function returns the same object that **Looper()** (inherited from the BHandler class) does—except that **Window()** returns it more specifically as a pointer to a BWindow and **Looper()** returns it more generally as a pointer to a BLooper.

See also: **BHandler::Looper()** in the Application Kit, **AddChild()**, **BWindow::AddChild()**, **AttachedToWindow()**

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_frame"	`B_RECT_TYPE`	The view's frame rectangle.
"_resize_mode"	`B_INT32_TYPE`	Resizing mode.
"_flags"	`B_INT32_TYPE`	View flags.
"_fname" (array)	`B_STRING_TYPE`	Font family and style.
"_fflt" (array)	`B_FLOAT_TYPE`	Font size, shear, and rotation (-1 if default).
"_color" (array)	`B_INT32_TYPE`	High, low, and view colors.
"_evmask" (array)	`B_INT32_TYPE`	Event mask and options.
"_dbuf"	`B_INT32_TYPE`	Double buffering flag.
"_origin"	`B_POINT_TYPE`	Origin.
"_psize"	`B_FLOAT_TYPE`	Pen size.
"_ploc"	`B_POINT_TYPE`	Pen location.
"_lmcapjoin" (array)	`B_INT16_TYPE`	Line cap mode and join mode.
"_lmmiter"	`B_FLOAT_TYPE`	Line mode miter value.
"_blend" (array)	`B_INT16_TYPE`	Blending alpha and mode.
"_dmod"	`B_INT32_TYPE`	Drawing mode.
"_views"	`B_MESSAGE_TYPE`	Archived child views (deep copy only).

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the font's family and style were not changed, the "_fname" field won't be found in the archive.

The Be Book

,

...in lovely HTML...

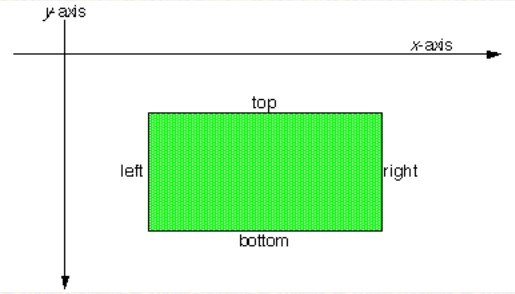
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRect

Derived from: *none*
Declared in: be/interface/Rect.h
Library: libbe.so
Allocation: Constructor or on the stack, typically the latter.
Summary: *more...*

A BRect object represents a rectangle. BRects are used throughout the Interface Kit to define the frames of windows, views, bitmaps—even the screen itself. A BRect is defined by its four sides, expressed as the public data members **left**, **top**, **right**, and **bottom**.



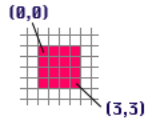
When used in the screen coordinate system(as a window or view's frame, for example) a BRect's sides are aligned with the x and y axes (as shown here), and its coordinate values, which are stored as floats, are floored.

Rectangle Size and Area

You would expect a BRect defined thus...:

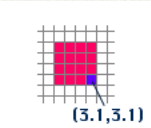
```
BRect rect(0, 0, 3, 3);
```

...to have a width of 3.0 and a height of 3.0. These, indeed, are the values returned by the **Width()** and **Height()** functions. However, the coordinate system considers integer coordinates to fall in the center of pixels, so the rectangle "touches" a 4x4 pixel grid when it's applied to the screen—it appears one pixel wider and one higher than **Width()** and **Height()** would have you believe. The mapping of rectangle coordinates to pixels is explained in greater detail in "The Coordinate Space".



A rectangle's area includes the points that lie along its sides, but it doesn't necessarily contain the entire area of the pixels that it "lights up." For example, consider the point at (3.1, 3.1).

This point falls outside the (0,0,3,3) BRect defined above (i.e the point doesn't **`Intersect()`** with the BRect), even though it corresponds to one of the pixels that the BRect touches (as shown here).



Rectangle Validity

To represent a valid rectangle, a BRect's **`top`** value must be less than or equal to **`bottom`**, and its **`left`** must be less than or equal to **`right`**. Invalid rectangles are meaningless and can't be used (to define a window or view's area, etc.) Note that the BRect constructor and Set...() function don't prevent you from creating an invalid rectangle. Use the **`IsValid()`** boolean function to test a BRect object's validity.

Data Members

float **`left`**
The value of the rectangle's left side.

float **`top`**

The value of the rectangle's top.

float **`right`**

The value of the rectangle's right side.

float **`bottom`**

The value of the rectangle's bottom.

Constructor

BRect()

```
inline BRect(float left, float top, float right, float bottom)
inline BRect(BPoint leftTop, BPoint rightBottom)
inline BRect(const BRect& rect)
inline BRect(void)
```

Initializes a BRect as four sides, as two diametrically opposed corners, or as a copy of some other BRect object. A rectangle that's not assigned any initial values is invalid, until a specific assignment is made, either through a `Set()` function or by setting the object's data members directly.

Member Functions

Contains() , Intersects()

bool **Contains(**BPoint *point***)** const bool **Contains(**BRect *rect***)** const
bool **Intersects(**BRect *rect***)** const

Contains() returns `true` if *point* or *rect* lies entirely within the BRect's rectangle (and **false** if not). A rectangle contains the points that lie along its edges; for example, two identical rectangles contain each other.

Intersect() returns `true` if the BRect has any area—even a corner or part of a side—in common with *rect*, and `false` if it doesn't.

This function's results are unpredictable if either rectangle is invalid.

See also: **&** (intersection), **|** (union), **BPoint::ConstrainTo()**

Height() see Width()

InsetBy() , InsetBySelf() , InsetByCopy() , OffsetBy() , OffsetBySelf() , OffsetByCopy() , OffsetTo() , OffsetToSelf() , OffsetToCopy()

void **InsetBy(**float *x*, float *y***)** void **InsetBy(**BPoint *point***)**
BRect &**InsetBySelf(**float *x*, float *y***)** BRect &**InsetBySelf(**BPoint *point***)**
BRect **InsetByCopy(**float *x*, float *y***)** BRect **InsetByCopy(**BPoint *point***)**
void **OffsetBy(**float *x*, float *y***)** void **OffsetBy(**BPoint *point***)**
BRect &**OffsetBySelf(**float *x*, float *y***)** BRect &**OffsetBySelf(**BPoint *point***)**
BRect **OffsetByCopy(**float *x*, float *y***)** BRect **OffsetByCopy(**BPoint *point***)**
void **OffsetTo(**float *x*, float *y***)** void **OffsetTo(**BPoint *point***)**
BRect &**OffsetToSelf(**float *x*, float *y***)** BRect &**OffsetToSelf(**BPoint *point***)**
BRect **OffsetToCopy(**float *x*, float *y***)**

BRect **OffsetToCopy(**BPoint *point***)**

Sorting out the different versions, there are three basic rectangle-manipulation functions here:

- **InsetBy()** insets the sides of the BRect's rectangle by *x* units (left and right sides) and *y* units (top and bottom). Positive inset values shrink the rectangle; negative values expand it. Note that both sides of each pair moves the full amount. For example, if you inset a BRect by (4,4), the left side moves (to the right) four units and the right side moves (to the left) four units (and similarly with the top and bottom).
- **OffsetBy()** moves the BRect horizontally by *x* units and vertically by *y* units. The rectangle's size doesn't change.
- **OffsetTo()** moves the BRect to the location (*x*,*y*).

If a BPoint argument is used, the BPoint's **x** and **y** values are used as the *x* and *y* arguments.

The **...Self()** versions of the functions are the same as the simpler versions, but they conveniently return the modified BRect. The **...Copy()** versions copy the BRect, and then modify and return the copy (without changing the original).

Intersects() see Contains()

IsValid()

inline bool **IsValid(**void**)** const

Returns **true** if the BRect's right side is greater than or equal to its left and its bottom is greater than or equal to its top, and `false` otherwise. An invalid rectangle can't be used to define an interface area (such as the frame of a view or window).

LeftBottom(), LeftTop() see Set()
OffsetBy(), OffsetByCopy(), OffsetBySelf() see InsetBy()
OffsetTo(), OffsetToCopy(), OffsetToSelf() see InsetBy()

PrintToStream()

void **PrintToStream(**void**)** const

Prints the contents of the BRect object to standard out in the form:

```
"BRect(left, top, right, bottom)"
```

RightBottom(), RightTop() see Set()

Set() , SetLeftTop() , SetLeftBottom() , SetRightTop() , SetRightBottom()

inline void **Set(**float *left*, float *top*, float *right*, float *bottom***)**
void **SetLeftTop(**const BPoint *point***)**
void **SetLeftBottom(**const BPoint *point***)**
void **SetRightTop(**const BPoint *point***)**
void **SetRightBottom(**const BPoint *point***)**
BPoint **LeftTop(**void**)** const
BPoint **LeftBottom(**void**)** const
BPoint **RightTop(**void**)** const
BPoint **RightBottom(**void**)** const

Set() sets the object's rectangle by defining the coordinates of all four sides. The other **Set...()** functions move one of the rectangle's corners to the BPoint argument; the other corners and sides are modified concomittantly. None of these functions prevents you from creating an invalid rectangle.

The BPoint-returning functions return the coordinates of one of the rectangle's four corners.

Width() , IntegerWidth() , Height() , IntegerHeight()

inline float **Width(**void**)** const
inline int32 **IntegerWidth(**void**)** const
inline float **Height(**void**)** const
inline int32 **IntegerHeight(**void**)** const

Width() returns the numerical difference between the rectangle's right and left sides (i.e. **right** – **left**). **IntegerWidth()** does the same, but rounds up in the case of a fractional difference.

Height() and **IntegerHeight()** perform similar calculations for the height of the rectangle (i.e. **bottom** – **top** and ceil(**bottom** – **top**)).

The width and height of a BRect's rectangle, as returned through these functions

---

Operators

= (assignment)

```
inline BRect& operator =(const BRect& from)
```

Copies *from*'s rectangle data into the left-side object.

== (equality), != (inequality)

```
bool operator ==(BRect) const
bool operator !=(BRect) const
```

== returns `true` if the two objects' rectangles exactly coincide.

!= returns `true` if the two objects' rectangles don't coincide.

& (intersection)

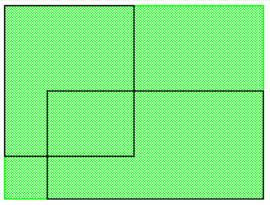
```
BRect operator &(BRect) const
```

Creates and returns a new BRect that's the intersection of the two operands. The new BRect encloses the area that the two operands have in common. If the two operands don't intersect, the new BRect will be invalid.

| (union)

```
BRect operator |(BRect) const
```

Creates and returns a new BRect that minimally but completely encloses the area defined by both of the operands. The shaded area illustrates the union of the two outlined rectangles:



The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRect Function Summary

Function Summary

Function	Description
!=	Compares the components of one BRect with another for inequality
&	Returns the intersection of two BRects
=	Copies the coordinates of one BRect to another
==	Compares the components of one BRect with another for equality
Contains()	Tests whether a BPoint or a BRect lies entirely within the rectangle
Height()	Returns width of the rectangle, rounded up to the next whole number
InsetBy()	Shrinks rectangle on all sides by given amounts
InsetByCopy()	Shrinks rectangle on all sides by given amounts
InsetBySelf()	Shrinks rectangle on all sides by given amounts
IntegerHeight()	Returns height of the rectangle, rounded up to the next whole number
IntegerWidth()	Returns the difference between the top and bottom coordinates
Intersects()	Tests whether a BRect intersects the rectangle
IsValid()	Tests whether the rectangle's coordinates

	are valid
OffsetBy()	Moves the rectangle by some amount
OffsetByCopy()	Moves the rectangle by some amount
OffsetBySelf()	Moves the rectangle by some amount
OffsetTo()	Moves the rectangle to another location
OffsetToCopy()	Moves the rectangle to another location
OffsetToSelf()	Moves the rectangle to another location
PrintToStream()	Prints the coordinates of the rectangle to standard output
BRect()	Constructor
Set()	Sets the rectangle
SetLeftBottom()	Sets the rectangle's bottom left corner
SetLeftTop()	Sets the rectangle's top left corner
SetRightBottom()	Sets the rectangle's bottom right corner
SetRightTop()	Sets the rectangle's top right corner
Width()	Returns the difference between the left and right coordinates
|	Returns the smallest rectangle enclosing two BRects

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

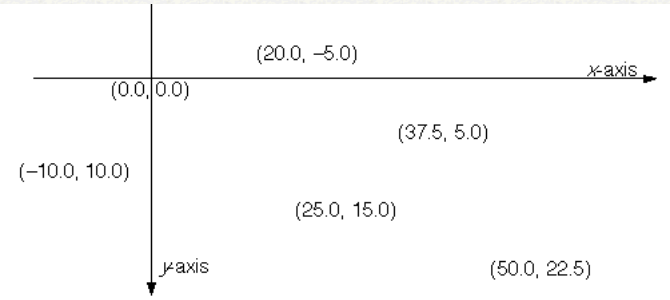
Copyright © 2000 Be, Inc. All rights reserved..

The Coordinate Space

To locate screens, windows, and views, draw in them, and report where the cursor is positioned, it's necessary to have some conventional way of talking about the display surface. The same conventions are used whether the display device is a monitor that shows images on a screen or a printer that puts them on a page.

Two-Dimensional Coordinates

In Be software, the display surface is described by a two-dimensional coordinate system where *y*-coordinate values are greater towards the bottom of the display and smaller towards the top, *x*-coordinate values are greater to the right and smaller to the left:



All quantities and distances are measured by floating-point values of the type `float`.

A single coordinate unit is 1/72 of an inch, roughly equal to a typographical point. However, all screens are considered to have a resolution of 72 pixels per inch (regardless of the actual dimension), so coordinate units actually count screen pixels. In other words, one unit is the distance between the centers of adjacent pixels on-screen.

Coordinate Systems

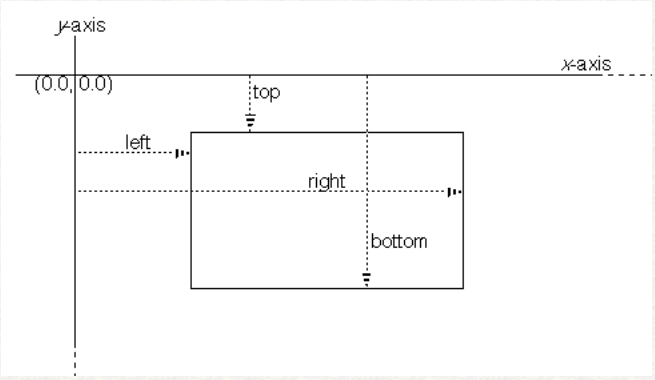
Specific coordinate systems are associated with the screen, with windows, and with the views inside windows. They differ only in where the two axes are located:

- The global or *screen coordinate system* has its origin, (0.0, 0.0), at the left top corner of the main screen. It's used for positioning windows on-screen, for arranging multiple screens connected to the same machine, and for comparing coordinate values that weren't originally stated in a common coordinate system. (Multiple screens are not currently supported.)
- A *window coordinate system* has its origin at the left top corner of the content area of a window. It's used principally for positioning views within the window. Each window has its own coordinate system so that locations within the window can be specified without regard to where the window happens to be on-screen.
- A *view coordinate system* has its default origin at the left top corner of the view rectangle. However, scrolling can shift view coordinates and move the origin. View-specific coordinates are used for all drawing operations and to report the cursor location in most system messages.

Coordinate Data Types

The Interface Kit defines a handful of basic classes for locating points and areas within a coordinate system:

- A BPoint object is the simplest way to specify a coordinate location. Each object stores two values—an *x*-coordinate and a *y*-coordinate—that together locate a specific point, (*x*, *y*), within a given coordinate system.
- A BRect object represents a rectangle; it's the simplest way to designate an area within a coordinate system. The BRect class defines a rectangle as a set of four coordinate values —corresponding to the rectangle's left, top, right, and bottom edges, as shown in this illustration:



- The sides of the rectangle are therefore parallel to the coordinate axes. The left and right sides delimit the range of *x*-coordinate values within the rectangle, and the top and bottom sides delimit the range of *y*-coordinate values. For example, if a rectangle's left top corner is at (0.8, 2.7) and its right bottom corner is at (11.3, 49.5), all points having *x*-coordinates ranging from 0.8 through 11.3 and *y*coordinates from 2.7 through 49.5 lie inside the rectangle.
- If the top of a rectangle is the same as its bottom, or its left the same as its right, the rectangle defines a straight line. If the top and bottom are the same and also the left and right, it collapses to a single point. Such rectangles are still valid—they specify real locations within a coordinate system. However, if the top is greater than the bottom or the left greater than the right, the rectangle is invalid; it has no meaning.
- A BPolygon object represents a polygon, a closed figure with an arbitrary number of sides. The polygon is defined as an ordered set of points. It encloses the area that would be outlined by connecting the points in order, then connecting the first and last points to close the figure. Each point is therefore a potential vertex of the polygon.
- A BRegion object defines a set of points. A region can be any shape and even include discontinuous areas. It's equivalent to a set of rectangles.

Mapping Coordinates to Pixels

The device-independent coordinate space described above must be mapped to the pixel grid of a particular display device—the screen, a printer, or some other piece of hardware that's capable of rendering an image. For example, to display a rectangle, it's necessary to find the pixel columns that correspond to its right and left sides and the pixel rows that correspond to its top and bottom.

This depends entirely on the resolution of the device. In essence, each device-independent coordinate value must be translated internally to a device-dependent value—an integer index to a particular column or row of pixels. In the coordinate space of the device, one unit equals one pixel.

This translation is easy for the screen, since, as mentioned above, there's a one-to-one correspondence between coordinate units and pixels. It reduces to rounding floating-point coordinates to integers. For other devices, however, the translation means first scaling the coordinate value to a device-specific value, then rounding. For example, the point (12.3, 40.8) would translate to (12, 41) on the screen, but to (51, 170) on a 300 dpi printer.

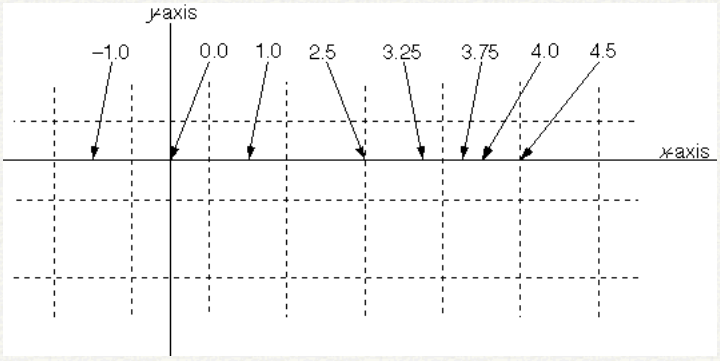
Screen Pixels

To map coordinate locations to device-specific pixels, you need to know only two things:

- The resolution of the device
- The location of the coordinate axes relative to pixel boundaries

The axes are located in the same place for all devices: The *x*-axis runs left to right along the middle of a row of pixels, and the *y*-axis runs down the middle of a pixel column. They meet at the very center of a pixel.

Because coordinate units match pixels on the screen, this means that all integral coordinate values (those without a fractional part) fall midway across a screen pixel. The following illustration shows where various *x*-coordinate values fall on the *x*-axis. The broken lines represent the division of the screen into a pixel grid:



As this illustration shows, it's possible to have coordinate values that lie on the boundary between two pixels. The next section describes how these values are mapped to one pixel or

the other.

Picking Pixels to Stroke and Fill

This section discusses how the various BView `Stroke...()` and `Fill...()` functions pick specific pixels to color. (Since pixels are small, this is not a topic that you necessarily need to be concerned about, especially as you begin programming for the BeOS.)

Pixels are chosen after the pen size and all coordinate values have been translated to device-specific units. Device-specific values measure distances by counting pixels; 1 unit equals 1 pixel on the device.

A device-specific value can be derived from a coordinate value using a formula that takes the size of a coordinate unit and the resolution of the device into account. For example:

```
device_value= coordinate_value*( dpi/ 72 )
```

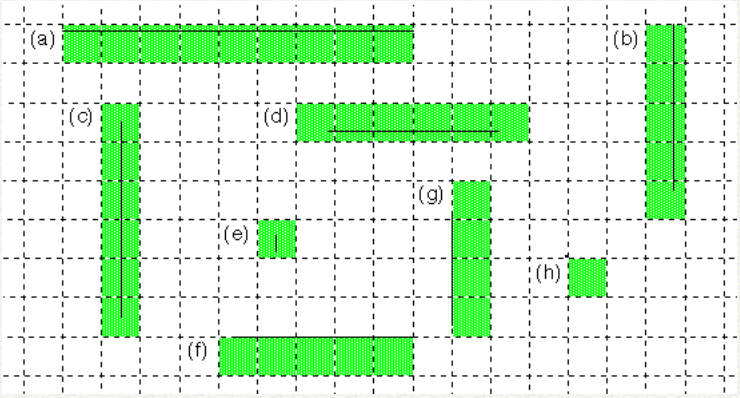
dpi is the resolution of the device in dots (pixels) per inch, 72 is the number of coordinate units in an inch, and *device_value* is rounded to the closest integer.

To describe where lines and shapes fall on the pixel grid, this section mostly talks about pixel units rather than coordinate units. The accompanying illustrations magnify the grid so that pixel boundaries are clear. As a consequence, they can show only very short lines and small shapes. By blowing up the image, they exaggerate the phenomena they illustrate.

Stroking Thin Lines

The thinnest possible line is drawn when the pen size translates to 1 pixel on the device. Setting the size to 0.0 coordinate units guarantees a 1-pixel pen on all devices.

A 1-pixel pen follows the path of the line it strokes and makes the line exactly 1 pixel thick at all points. If the line is perfectly horizontal or vertical, it touches just one row or one column of pixels, as illustrated below. (The grid of broken lines shows the separation of the display surface into pixels.)

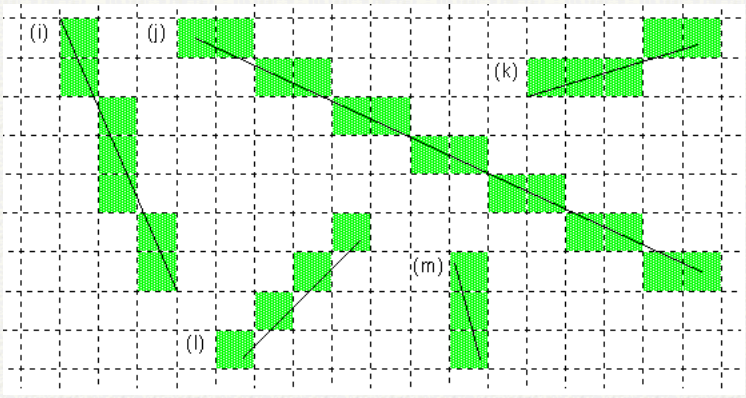


Only pixels that the line path actually passes through are colored to display the line. If a path begins or ends on a pixel boundary, as it does for examples (a) and (b) in the illustration, the

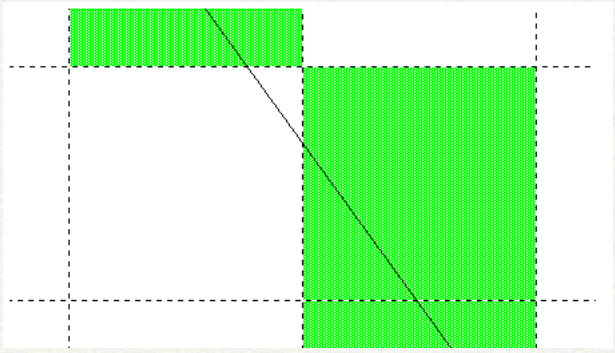
pixels at the boundary aren't colored unless the path crosses into the pixel. The pen touches the fewest possible number of pixels.

A line path that doesn't enter any pixels, but lies entirely on the boundaries between pixels, colors the pixel row beneath it or the pixel column to its right, as illustrated by lines (f) and (g). A path that reduces to a single point lying on the corner of four pixels, as does (h), colors the pixel at its lower right. (However, currently, it's indeterminate which column or row of adjacent pixels would be used to display vertical and horizontal lines like (f) and (g). Point (h) would not be visible.)

1-pixel lines that aren't exactly vertical or horizontal touch just one pixel per row or one per column. If the line is more vertical than horizontal, only one pixel in each row is used to color the line. If the line is more horizontal than vertical, only one pixel in each column is used. Some illustrations of slanted 1-pixel thick lines are given below:

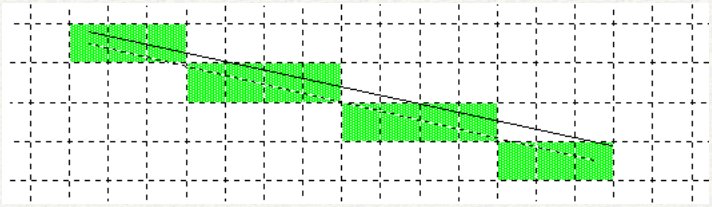


Although a 1-pixel pen touches only pixels that lie on the path it strokes, it won't touch every pixel that the path crosses if that would mean making the line thicker than specified. When the path cuts though two pixels in a column or row, but only one of those pixels can be colored, the one that contains more of the path (the one that contains the midpoint of the segment cut by the column or row) is chosen. This is illustrated in the following close-up, which shows where a mostly vertical line crosses one row of pixels:



However, before a choice is made as to which pixel in a row or column to color, the line path is normalized for the device. For example, if a line is defined by two endpoints, it's first determined which pixels correspond to those endpoints. The line path is then treated as if it connected the centers of those pixels. This may alter which pixels get colored, as is illustrated

below. In this illustration, the solid black line is the line path as originally specified and the broken line is its normalized version:



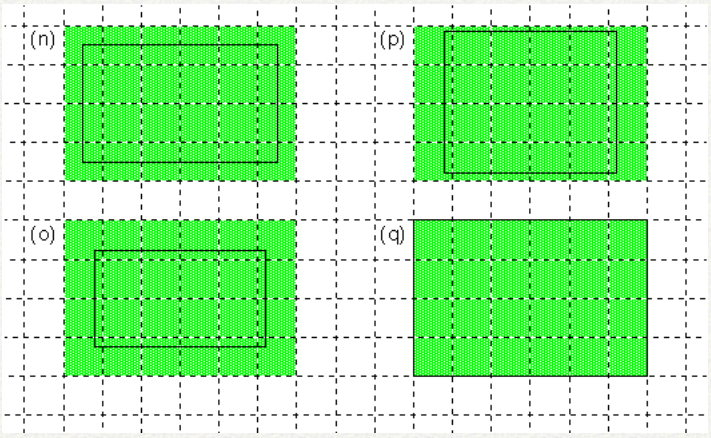
This normalization is nothing more than the natural consequence of the rounding that occurs when coordinate values are translated to device-specific pixel values.

Stroking Curved Lines

Although all the diagrams above show straight lines, the principles they illustrate apply equally to curved line paths. A curved path can be treated as if it were made up of a large number of short straight segments.

Filling and Stroking Rectangles

The following illustration shows how some rectangles, represented by the solid black line, would be filled with a solid color.



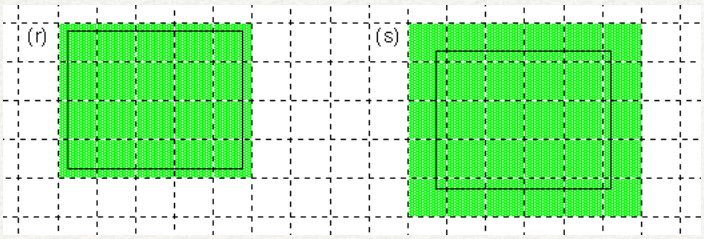
A rectangle includes every pixel that it encloses and every pixel that its sides pass through. However, as rectangle (q) illustrates, it doesn't include pixels that its sides merely touch at the boundary.

If the pixel grid in this illustration represents the screen, rectangle (q) would have left, top, right, and bottom coordinates with fractional values of .5. Rectangle (n), on the other hand, would have coordinates without any fractional parts. Nonfractional coordinates lie at the center of screen pixels.

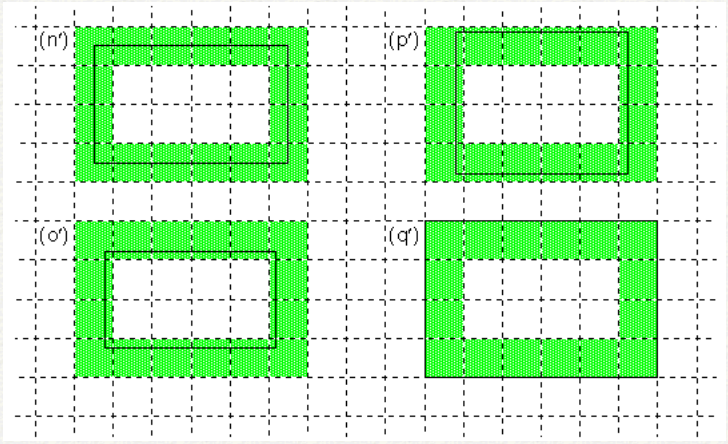
Rectangle (n), in fact, is the normalized version of all four of the illustrated rectangles. It shows how the sides of the four rectangles would be translated to pixel values. Note that for a rectangle like (q), with edges that fall on pixel boundaries, normalization means rounding the

left and top sides upward and rounding the right and bottom sides downward. This follows from the principal that the fewest possible number of pixels should be colored.

Although the four rectangles above differ in size and shape, when filled they all cover a 6 * 4 pixel area. You can't predict this area from the dimensions of the rectangle. Because the coordinate space is continuous and *x* and *y* values can be located anywhere, rectangles with different dimensions might have the same rendered size, as shown above, and rectangles with the same dimensions might have different rendered sizes, as shown below:



If a one-pixel pen strokes a rectangular path, it touches only pixels that would be included if the rectangle were filled. The illustration below shows the same rectangles that were presented above, but strokes them rather than fills them:

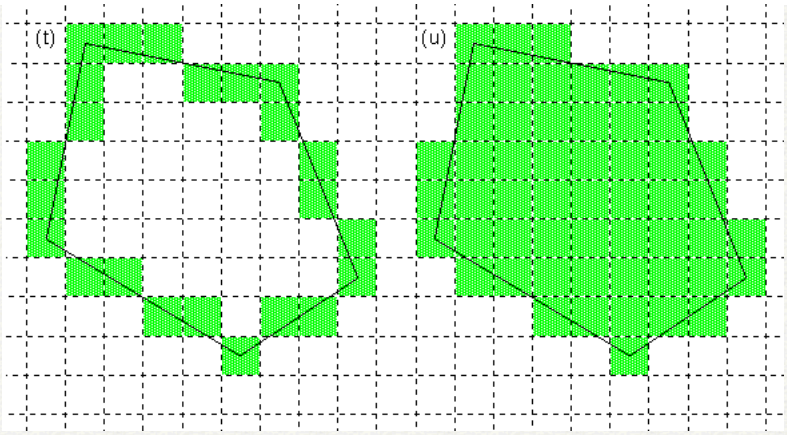


Each of the rectangles still covers a 6 * 4 pixel area. Note that even though the path of rectangle (q) lies entirely on pixel boundaries, pixels below it and to its right are not touched by the pen. The pen touches only pixels that lie within the rectangle.

If a rectangle collapses to a straight line or to a single point, it no longer contains any area. Stroking or filling such a rectangle is equivalent to stroking the line path with a one-pixel pen, as was discussed in the previous section.

Filling and Stroking Polygons

The figure below shows a polygon as it would be stroked by a one-pixel pen and as it would be filled:

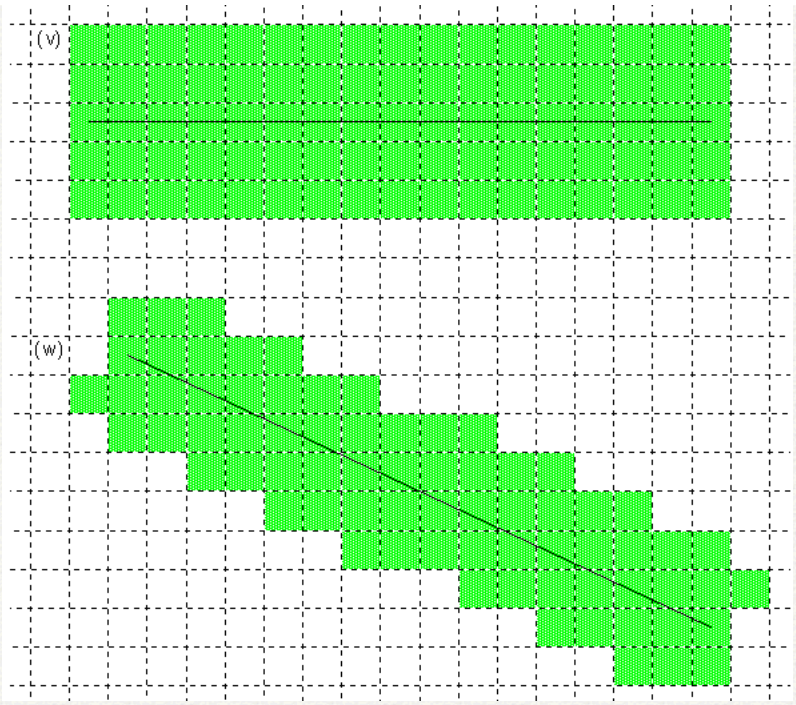


The same rules apply when stroking each segment of a polygon as would apply if that segment were an independent line. Therefore, the pen may not touch every pixel the segment passes through.

When the polygon is filled, no additional pixels around its border are colored. As is the case for a rectangle, the displayed shape of filled polygon is identical to the shape of the polygon when stroked with a one-pixel pen. The pen doesn't touch any pixels when stroking the polygon that aren't colored when the polygon is filled. Conversely, filling doesn't color any pixels at the border of the polygon that aren't touched by a one-pixel pen.

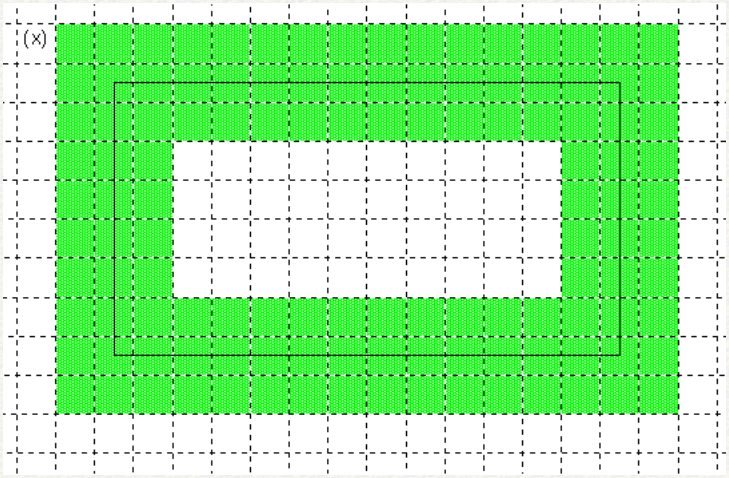
Stroking Thick Lines

A pen that's thicker than one pixel touches the same pixels that a one-pixel pen does, but it adds extra columns and rows adjacent to the line path. A thick pen tip is, in effect, a linear brush that's held perpendicular to the line path and kept centered on the line. The illustration below shows two short lines, each five pixels thick:



The thickness or a vertical or horizontal line can be measured in an exact number of pixels. When the line is slanted, as it is for (t) above, the stroking algorithm tries to make the line visually approximate the thickness of a vertical or horizontal line. In this way, lines retain their shape even when rotated.

When a rectangle is stroked with a thick pen, the corners of the rectangle are filled in, as shown in the example below:



The Be Book The Interface Kit The Interface Kit Index

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPoint

Derived from: *none*
Declared in: be/interface/Point.h
Library: libbe.so
Allocation: Constructor or on the stack, typically the latter.
Summary: *more...*

BPoint objects represent points on a two-dimensional coordinate grid. The object's coordinates are stored as public **x** and **y** data members. The BPoint class declares a global object, **B_ORIGIN**, that represents (0,0).

For an overview of coordinate geometry for the BeOS, see "The Coordinate Space".

Data Members

float **x**

The coordinate value measured horizontally along the *x*-axis.

float **y**

The coordinate value measured vertically along the *y*-axis.

Constructor

BPoint()

```
inline BPoint(float x, float y)
inline BPoint(const BPoint& point)
inline BPoint(void)
```

Creates a new BPoint object that corresponds to the point (*x*, *y*), or that's copied from *point*. If no coordinate values are assigned, the BPoint's location is indeterminate.

See also: `Set()`, the assignment operator

Member Functions

ConstrainTo()

```
void ConstrainTo(BRect rect)
```

Ensures that the BPoint lies within *rect*. If it's already contained in the rectangle, the BPoint is unchanged; otherwise, it's moved to the rect's nearest edge.

See also: `BRect::Contains()`

PrintToStream()

```
void PrintToStream(void) const
```

Prints the BPoint's coordinates to standard output in the form:

```
"BPoint(x, y)"
```

Set()

```
inline void Set(float x, float y)
```

Sets the BPoint's x and y coordinates.

See also: the BPoint constructor

Operators

= (assignment)

```
inline BPoint& operator =(const BPoint &from)
```

Copies *from*'s coordinate data into the left-side object.

== (equality), != (inequality)

```
bool operator ==(BPoint) const
bool operator !=(BPoint) const
```

== returns **true** if the two objects' point exactly coincide.

!= returns **true** if the two objects' points don't coincide.

+ (addition), += (addition and assignment)

```
BPoint operator +(const BPoint&) const
BPoint& operator +=(const BPoint&)
```

+ creates and returns a new BPoint that adds the two operands together. The new object's **x** coordinate is the sum of the operands' **x** values; its **y** value is the sum of the operands' **y** values.

+= adds the operands together and stores the result in the left operand.

(subtraction), = (subtraction and assignment)

```
BPoint operator -(const BPoint&) const
BPoint& operator -=(const BPoint&)
```

- creates and returns a new BPoint that subtracts the right operand from the left. The new object's **x** coordinate is the difference between the operands' **x** values; its **y** value is the difference between the operands' **y** values.

-= performs the subtraction and stores the result in the left operand.

Global Objects

B_ORIGIN

BPoint object that represents (0,0).

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPoint Function Summary

Function Summary

Function	Description
!=	Returns true if the operands don't coincide
+	Component-wise addition of two BPoints
+=	Component-wise addition of two BPoints
=	Component-wise subtraction of one BPoint from another
==	Returns true if the operands coincide
ConstrainTo()	Constrains point to lie within a BRect
BPoint()	Constructor
PrintToStream()	Outputs contents of BPoint to standard output
Set()	Assigns coordinates to the BPoint

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPolygon

Derived from: *none*
Declared in: be/interface/Polygon.h
Library: libbe.so
Summary: *more...*

A BPolygon object represents a *polygon*—a closed, many-sided figure that describes an area within a two-dimensional coordinate system. It differs from a BRect object in that it can have any number of sides and the sides don't have to be aligned with the coordinate axes.

A BPolygon is defined as a series of connected points. Each point is a potential vertex in the polygon. An outline of the polygon could be constructed by tracing a straight line from the first point to the second, from the second point to the third, and so on through the whole series, then by connecting the first and last points if they're not identical.

The BView functions that draw a polygon—`StrokePolygon()` and `FillPolygon()`—take BPolygon objects as arguments. `StrokePolygon()` offers the option of leaving the polygon open—of not stroking the line that connects the first and last points in the list. The polygon therefore won't look like a polygon, but like an chain of lines fastened at their endpoints.

Constructor and Destructor

BPolygon()

```
BPolygon(const BPoint *pointList, int32 numPoints)
BPolygon(const BPolygon *polygon)
BPolygon(void)
```

Initializes the BPolygon by copying *numPoints* from *pointList*, or by copying the list of points from another *polygon*. If one polygon is constructed from another, the original and the copy won't share any data; independent memory is allocated for the copy to hold a duplicate list of points.

If a BPolygon is constructed without a point list, points must be set with the **AddPoints()** function.

See also: **AddPoints()**

~BPolygon()

```
virtual ~BPolygon()
```

Frees all the memory allocated to hold the list of points.

Member Functions

AddPoints()

void **AddPoints(**const BPoint **pointList*, int32 *numPoints***)**

Appends *numPoints* from *pointList* to the list of points that already define the polygon.

See also: the BPolygon constructor

CountPoints()

int32 **CountPoints(**void**)** const

Returns the number of points that define the polygon.

Frame()

BRect **Frame(**void**)** const

Returns the polygon's frame rectangle—the smallest rectangle that encloses the entire polygon.

MapTo()

void **MapTo(**BRect *source*, BRect *destination***)**

Modifies the polygon so that it fits the *destination* rectangle exactly as it originally fit the *source* rectangle. Each vertex of the polygon is modified so that it has the same proportional position relative to the sides of the destination rectangle as it originally had to the sides of the source rectangle.

The polygon doesn't have to be contained in either rectangle. However, to modify a polygon so that it's exactly inscribed in the destination rectangle, you should pass its frame rectangle as the source:

```
BRect frame = myPolygon->Frame();
myPolygon->MapTo(frame, anotherRect);
```

PrintToStream()

void **PrintToStream(**void**)** const

Prints the BPolygon's point list to the standard output stream (`stdout`). The BPoint version of this function is called to report each point as a string in the form

```
"BPoint(x, y)"
```

where *x* and *y* stand for the coordinate values of the point in question.

See also: **`BPoint::PrintToStream()`**

Operators

= (assignment)

```
BPolygon& operator =(const BPolygon&)
```

Copies the point list of one BPolygon object and assigns it to another BPolygon. After the assignment, the two objects describe the same polygon, but are independent of each other. Destroying one of the objects won't affect the other.

The Be Book The Interface Kit The Interface Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPolygon Function Summary

Function Summary

Function	Description
=	Copies one BPolygon to another
AddPoints()	Appends points to current list of points
CountPoints()	Returns number of points in the polygon
Frame()	Returns smallest rectangle enclosing the polygon
MapTo()	Scales and translates object from one BRect to another
BPolygon()	Constructor
~BPolygon()	Destructor
PrintToStream()	Outputs contents of BPolygon to standard output

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Drawing-Related Functions

Derived from: BHandler
Declared in: be/interface/View.h
Library: libbe.so
Summary: *more...*

The primitive drawing functions are listed in the first part of this section. Other drawing functions (which can be used to create a reusable sequence of primitive drawing functions, to install a default background bitmap into a view, to force drawing or redrawing, and for a few other purposes) are listed immediately following the primitive drawing functions, in Other Drawing Functions

---

Primitive Drawing Functions

Below are the primitive drawing functions (those which draw circles, lines, etc.) found in the BView class.

DrawBitmap() , DrawBitmapAsync()

```
void DrawBitmap(const BBitmap *image)
void DrawBitmap(const BBitmap *image, BPoint point)
void DrawBitmap(const BBitmap *image, BRect destination)
void DrawBitmap(const BBitmap *image, BRect source, BRect destination)
```

```
void DrawBitmapAsync(const BBitmap *image)
void DrawBitmapAsync(const BBitmap *image, BPoint point)
void DrawBitmapAsync(const BBitmap *image, BRect destination)
void DrawBitmapAsync(const BBitmap *image, BRect source, BRect destination)
```

These functions place a bitmap *image* in the view at the current pen position, at the *point* specified, or within the designated *destination* rectangle. The *point* and the *destination* rectangle are stated in the BView's coordinate system.

If a *source* rectangle is given, only that part of the bitmap image is drawn. Otherwise, the entire bitmap is placed in the view. The *source* rectangle is stated in the internal coordinates of the BBitmap object.

If the source image is bigger than the destination rectangle, it's scaled to fit.

The two functions differ in only one respect: `DrawBitmap()` waits for the Application Server to finish rendering the image before it returns. `DrawBitmapAsync()` doesn't wait; it passes the image to the server and returns immediately. The latter function can be more efficient in some cases—for example, you might use an asynchronous function to draw several bitmaps and then call `Sync()` to wait for them all to finish rather than wait for each one individually:

```
DrawBitmapAsync(bitmapOne, firstPoint);
DrawBitmapAsync(bitmapTwo, secondPoint);
DrawBitmapAsync(bitmapThree, thirdPoint);
Sync();
```

Or, if you can cram some useful work between the time you send the bitmap to the Application Server and the time you need to be sure that it has appeared on-screen, `DrawBitmapAsync()` will free your thread to do that work immediately:

```
DrawBitmapAsync(someBitmap, somePoint);
/* do something else */
Sync();
```

See also: the "Drawing" section near the beginning of this chapter, the BBitmap class, **Sync()**

DrawChar()

```
void DrawChar(char c)
void DrawChar(char c, BPoint point)
```

Draws the character *c* at the current pen position—or at the *point* specified—and moves the pen to a position immediately to the right of the character. This function is equivalent to passing a string of one character to **DrawString()**. The *point* is specified in the BView's coordinate system.

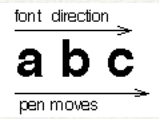
See also: **DrawString()**

DrawString()

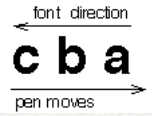
```
void DrawString(const char *string,
    escapement_delta *delta = NULL)
void DrawString(const char *string, int32 length,
    escapement_delta *delta = NULL)
void DrawString(const char *string, BPoint point,
    escapement_delta *delta = NULL)
void DrawString(const char *string, int32 length, BPoint point,
    escapement_delta *delta = NULL)
```

Draws the characters encoded in *length* bytes of *string*—or, if the number of bytes isn't specified, all the characters in the string, up to the null terminator ('0'). Characters are drawn in the BView's current font. The font's direction determines whether the string is drawn left-to-right or right-to-left. Its rotation determines the angle of the baseline (horizontal for an unrotated font). The spacing mode of the font determines how characters are positioned within the string and the string width.

This function places the characters on a baseline that begins one pixel above the current pen position—or one pixel above the specified *point* in the BView's coordinate system. It draws the characters to the right (assuming an unrotated font) and moves the pen to the baseline immediately past the characters drawn. For a left-to-right font, the pen will be in position to draw the next character, as shown below:



The characters are drawn in the opposite direction for a right-to-left font, but the pen still moves left-to-right:



The BeOS draws text one pixel above the logical baseline to maintain compatibility with an earlier version of one of our most commonly-used font rasterizers. This affects both fonts and BShapes representing glyphs (see **`BFont::GetGlyphShapes()`**. To draw text at the right place, add one to the Y coordinate when calling **`MovePenTo()`** or specifying a BPoint at which to begin drawing.

For a font that's read from left-to-right, a series of simple **`DrawString()`** calls (with no *point* specified) will produce a continuous string. For example, these two lines of code,

```
DrawString("tog");
DrawString("ether");
```

will produce the same result as this one,

```
DrawString("together");
```

except if the spacing mode is **`B_STRING_SPACING`**. Under **`B_STRING_SPACING`**, character placements are adjusted keeping the string width constant. The adjustments are contextually dependent on the string and may therefore differ depending on whether there are two strings ("tog" and "ether") or just one ("together").

If a *delta* argument is provided, **`DrawString()`** adds the additional amounts specified in the **`escapement_delta`** structure to the width of each character. This structure has two fields:

float **nonspace**
The amount to add to the width of characters that have visible glyphs (that put ink on the printed page).

float **space**
The amount to add to the width of characters that have escapements, but don't have visible glyphs (characters that affect the position of surrounding characters but don't put ink on the page).

When drawing to the screen, **`DrawString()`** uses antialiasing—unless the BView's font disables it or the font size is large enough (over 1,000.0 points) so that its benefits aren't required. Antialiasing produces colors at the margins of character outlines that are intermediate between the color of the text (the BView's high color) and the color of the background against which the text is drawn. When drawing in **`B_OP_COPY`** mode, antialiasing requires the BView's low color to match the background color.

It's much faster to draw a string in **`B_OP_COPY`** mode than in any other mode. If you draw the same string repeatedly in the same location in **`B_OP_OVER`** mode without erasing, antialiasing will produce different, and worse, results each time as the intermediate color it previously

produced is treated as the new background each time. Antialiasing doesn't produce pleasing results in **B_OP_SELECT** mode.

This is a graphical drawing function, so any character that doesn't have an escapement or a visible representation (including white space) is replaced by an undefined character that can be drawn (currently an empty box). This includes all control characters (those with values less than **B_SPACE**, 0x20).

DrawString() doesn't erase before drawing.

See also: **MovePenBy()**, **SetFontName()**, the BFont class

---

- FillArc() see StrokeEllipse()
- FillBezier() see StrokeBezier()
- FillEllipse() see StrokeEllipse()
- FillPolygon() see StrokePolygon()
- FillRect() see StrokeRect()

- FillRegion()

void **FillRegion(**BRegion ***region**, pattern *aPattern* = B_SOLID_HIGH**)** const

Fills the *region* with the pattern specified by *aPattern*—or, if no pattern is specified, with the current high color. Filling a region is equivalent to filling all the rectangles that define the region.

See also: the BRegion class

- FillRoundRect() see StrokeRoundRect()
- FillTriangle() see StrokeTriangle()
- StrokeArc() see StrokeEllipse()

- StrokeBezier() , FillBezier()

void **StrokeBezier(**BPoint ***controlPoints**, pattern *aPattern* = B_SOLID_HIGH**)**
void **FillBezier(**BPoint ***controlPoints**, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a third degree Bezier curve. **StrokeBezier()** strokes a line along the path of the curve; the width of the line is determined by the current pen size. **FillBezier()** fills in the region defined by the path of the curve and the line joining the two endpoints.

controlPoints points to an array of the four points for the curve. Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither

function alters the current pen position.

See also: `SetPenSize()`, `StrokeRoundRect()`

See also: `StrokeEllipse()`

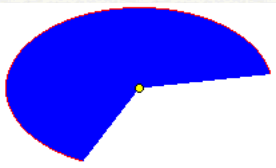
StrokeEllipse() , FillEllipse() , StrokeArc() , FillArc()

void **StrokeEllipse(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)** void **StrokeEllipse(**BPoint *center*, float *xRadius*, float *yRadius*, pattern *aPattern* = B_SOLID_HIGH
void **FillEllipse(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)** void **FillEllipse(**BPoint *center*, float *xRadius*, float *yRadius*, pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokeArc(**BRect *rect*, float *angle*, float *span*, pattern *aPattern* = B_SOLID_HIGH**)** void **StrokeArc(**BPoint *center*, float *xRadius*, float *yRadius*, float *angle*, float *span*, pattern *aPattern* = B_SOLID_HIGH**)**
void **FillArc(**BRect *rect*, float *angle*, float *span*, pattern *aPattern* = B_SOLID_HIGH**)** void **FillArc(**BPoint *center*, float *xRadius*, float *yRadius*, float *angle*, float *span*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw all or part of the ellipse that's inscribed in *rect* or that has its center at *center* and has horizontal and vertical radii *xRadius* and *yRadius*. The ellipse is always aligned with the x and y axes. A more flexible curve-drawing mechanism is given by `StrokeBezier()` and `FillBezier()`.

`StrokeEllipse()` strokes a line around the entire perimeter of the ellipse and `FillEllipse()` fills the area the ellipse encloses.

`StrokeArc()` and `FillArc()` stroke and fill a section of the ellipse, starting at *angle* (where 0 ° points right along the x-axis) and proceeding (counterclockwise) *span* degrees. In the illustration below, the red arc is the result of `StrokeArc()` with an angle of 10 ° and span of 235 °; the blue area is the same arc filled through `FillArc()`. The center of the ellipse (the yellow dot) is drawn for reference.



Currently, *angle* and *span* measurements in fractions of a degree are not supported.

For the stroking functions, the width of the stroked line is determined by the current pen size. All functions draw using *aPattern* or, if no pattern is specified, the current high color. The functions neither depend on nor alter the current pen position.

StrokeLine()

```
void StrokeLine(BPoint start, BPoint end, pattern aPattern = B_SOLID_HIGH)
void StrokeLine(BPoint end, pattern aPattern = B_SOLID_HIGH)
```

Draws a straight line between the *start* and *end* points—or, if no starting point is given, between the current pen position and *end* point—and leaves the pen at the end point.

This function draws the line using the current pen size and the specified pattern. If no pattern is specified, the line is drawn in the current high color. The points are specified in the BView's coordinate system.

See also: `SetPenSize()`, `BeginLineArray()`

StrokePolygon() , FillPolygon()

```
void StrokePolygon(BPolygon *polygon,
    bool isClosed = true, pattern aPattern = B_SOLID_HIGH)
void StrokePolygon(BPoint *pointList, int32 numPoints,
    bool isClosed = true, pattern aPattern = B_SOLID_HIGH)
void StrokePolygon(BPoint *pointList, int32 numPoints, BRect rect, bool isClosed = true,
pattern aPattern = B_SOLID_HIGH)
```

```
void FillPolygon(BPolygon *aPolygon,
    pattern aPattern = B_SOLID_HIGH)
void FillPolygon(BPoint *pointList, int32 numPoints, pattern aPattern = B_SOLID_HIGH)
void FillPolygon(BPoint *pointList, int32 numPoints, BRect rect, pattern aPattern = B_SOLID_HIG
```

These functions draw a polygon with an arbitrary number of sides. `StrokePolygon()` strokes a line around the edge of the polygon using the current pen size. If a *pointList* is specified rather than a BPolygon object, this function strokes a line from point to point, connecting the first and last points if they aren't identical. However, if the *isClosed* flag is `false`, `StrokePolygon()` won't stroke the line connecting the first and last points that define the BPolygon (or the first and last points in the *pointList*). This leaves the polygon open—making it not appear to be a polygon at all, but rather a series of straight lines connected at their end points. If *isClosed* is `true`, as it is by default, the polygon will appear to be a polygon, a closed figure.

`FillPolygon()` is a simpler function; it fills in the entire area enclosed by the polygon.

Both functions must calculate the frame rectangle of a polygon constructed from a point list—that is, the smallest rectangle that contains all the points in the polygon. If you know what this rectangle is, you can make the function somewhat more efficient by passing it as the *rect* parameter.

Both functions draw using the specified pattern—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: `SetPenSize()`, the BPolygon class

StrokeRect() , FillRect()

void **StrokeRect(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**
void **FillRect(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a rectangle. **StrokeRect()** strokes a line around the edge of the rectangle; the width of the line is determined by the current pen size. **FillRect()** fills in the entire rectangle.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**, **StrokeRoundRect()**

StrokeRoundRect() , FillRoundRect()

void **StrokeRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*, pattern *aPattern* = B_SOLID_HIGH
void **FillRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a rectangle with rounded corners. The corner arc is one-quarter of an ellipse, where the ellipse would have a horizontal radius equal to *xRadius* and a vertical radius equal to *yRadius*.

Except for the rounded corners of the rectangle, these functions work exactly like **StrokeRect()** and **FillRect()**.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **StrokeRect()**, **StrokeEllipse()**

StrokeShape() , FillShape()

void **StrokeShape(**BShape **shape*, pattern *aPattern* = B_SOLID_HIGH**)**
void **FillShape(**BShape **shape*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a shape. **StrokeShape()** strokes a line around the edge of the shape; the width of the line is determined by the current pen size. **FillShape()** fills in the entire shape.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**

StrokeTriangle() , FillTriangle()

void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,

pattern *aPattern* = B_SOLID_HIGH)
void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*, BRect *rect*,
pattern *aPattern* = B_SOLID_HIGH)

void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
pattern *aPattern* = B_SOLID_HIGH)
void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*, BRect *rect*,
pattern *aPattern* = B_SOLID_HIGH)

These functions draw a triangle, a three-sided polygon. `StrokeTriangle()` strokes a line the width of the current pen size from the first point to the second, from the second point to the third, then back to the first point. `FillTriangle()` fills in the area that the three points enclose.

Each function must calculate the smallest rectangle that contains the triangle. If you know what this rectangle is, you can make the function marginally more efficient by passing it as the *rect* parameter.

Both functions do their drawing using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: `SetPenSize()`

Other Drawing Functions

The functions below are BView drawing-related functions which do not fall under the rubric of Primitive Drawing Functions. The functions in this section can be used to draw bitmaps, to define sequences of more primitive drawing commands, and for some other more specialized purposes.

- AddLine() see BeginLineArray()
- AppendToPicture() see BeginPicture()

BeginLineArray() , AddLine() , EndLineArray()

void **BeginLineArray(**int32 *count***)**
void **AddLine(**BPoint *start*, BPoint *end*, rgb_color *color***)**
void **EndLineArray(**void**)**

These functions provide a more efficient way of drawing a large number of lines than repeated calls to **StrokeLine()**. **BeginLineArray()** signals the beginning of a series of up to *count* **AddLine()** calls; **EndLineArray()** signals the end of the series. Each **AddLine()** call defines a line from the *start* point to the *end* point, associates it with a particular *color*, and adds it to the array. The lines can each be a different color; they don't have to be contiguous. When **EndLineArray()** is called, all the lines are drawn—using the then current pen size—in the order that they were added to the array.

These functions don't change any graphics parameters. For example, they don't move the pen or change the current high and low colors. Parameter values that are in effect when **EndLineArray()** is called are the ones used to draw the lines. The high and low colors are ignored in favor of the *color* specified for each line.

The *count* passed to **BeginLineArray()** is an upper limit on the number of lines that can be drawn. Keeping the count close to accurate and within reasonable bounds helps the efficiency of the line-array mechanism. It's a good idea to keep it less than 256; above that number, memory requirements begin to impinge on performance.

See also: **StrokeLine()**

BeginPicture() , AppendToPicture() , EndPicture()

void **BeginPicture(**BPicture **picture***)**
void **AppendToPicture(**BPicture **picture***)**
BPicture ***EndPicture(**void**)**

BeginPicture() starts a new "picture recording" session: Subsequent drawing instructions invoked upon the view are recorded in the BPicture argument. `AppendToPicture()` does the same, but doesn't clear the argument first—it tacks additional instructions on to the end of the BPicture. **EndPicture()** ends the recording session; it returns the object that was passed to **BeginPicture() or AppendToPicture()**.

While it's recording a picture, the BView doesn't display anything to the screen. To render the drawing, you use the **DrawPicture()** function.

The picture captures only primitive graphics operations such as **DrawString()**, **FillArc()**, and **SetFont()**. Furthermore, only instructions performed by this view are recorded; the drawing done by the view's children is not recorded.

A BPicture can be recorded only if the BView is attached to a window. The window can be off-screen and the view itself can be hidden or reside outside the current clipping region.

See also: the BPicture class, **DrawPicture()**

ClearViewBitmap() see SetViewBitmap()

ClearViewOverlay() see SetViewOverlay()

CopyBits()

void **CopyBits(**BRect *source*, BRect *destination***)**

Copies the image displayed in the *source* rectangle to the *destination* rectangle, where both rectangles lie within the view and are stated in the BView's coordinate system.

If the two rectangles aren't the same size, the source image is scaled to fit.

If not all of the destination rectangle lies within the BView's visible region, the source image is clipped rather than scaled.

If not all of the source rectangle lies within the BView's visible region, only the visible portion is copied. It's mapped to the corresponding portion of the destination rectangle. The BView is then invalidated so its **Draw()** function will be called to update the part of the destination rectangle that can't be filled with the source image.

The BView must be attached to a window.

DrawPicture() , DrawPictureAsync()

void **DrawPicture(**const BPicture ***picture***)**
void **DrawPicture(**const BPicture ****picture***, BPoint *point***)**
void **DrawPicture(**const char ****filename***, off_t *offset*, BPoint *point***)**

void **DrawPictureAsync(**const BPicture ****picture***)**

void **DrawPictureAsync**(const BPicture **picture*, BPoint *point*)
void **DrawPictureAsync**(const char **filename*, long *offset*, BPoint *point*)

Draws the previously recorded *picture* at the current pen position—or at the specified *point* in the BView's coordinate system. The point or pen position is taken as the coordinate origin for all the drawing instructions recorded in the BPicture. The last form of the method plays a picture from an arbitrary offset of a file.

The two functions differ in only one respect: **DrawPicture()** waits for the Application Server to finish rendering the image before it returns. **DrawPictureAsync()** doesn't wait; it passes the image to the server and returns immediately. The latter function can be more efficient in some cases—for example, you might use an asynchronous function to draw several bitmaps and then call **Sync()** to wait for them all to finish rather than wait for each one individually:

Nothing that's done in the BPicture can affect anything in the BView's graphics state—for example, the BPicture can't reset the current high color or the pen position. Conversely, nothing in the BView's current graphics state affects the drawing instructions captured in the picture. The graphics parameters that were in effect when the picture was recorded determine what the picture looks like.

See also: **BeginPicture()**, the BPicture class

- EndLineArray() see BeginLineArray()
- EndPicture() see AppendToPicture()

- Flush() , Sync()

void **Flush**(void) const
void **Sync**(void) const

These functions flush the window's connection to the Application Server. If the BView isn't attached to a window, `Flush()` does nothing.

If the BView isn't attached to a window, `Sync()` will crash the application.

For reasons of efficiency, the window's connection to the Application Server is buffered. Drawing instructions destined for the server are placed in the buffer and dispatched as a group when the buffer becomes full. Flushing empties the buffer, sending whatever it contains to the server, even if it's not yet full.

The buffer is automatically flushed on every update. However, if you do any drawing outside the update mechanism—in response to interface messages, for example—you need to explicitly flush the connection so that drawing instructions won't languish in the buffer while waiting for it to fill up or for the next update. You should also flush it if you call any drawing functions from outside the window's thread.

Flush() simply flushes the buffer and returns. It does the same work as BWindow's function of the same name.

Sync() flushes the connection, then waits until the server has executed the last instruction that was in the buffer before returning. This alternative to **Flush()** prevents the application from getting ahead of the server (ahead of what the user sees on-screen) and keeps both processes synchronized.

It's a good idea, for example, to call **Sync()**, rather than **Flush()**, after employing BViews to produce a bitmap image (a BBitmap object). **Sync()** is the only way you can be sure the image has been completely rendered before you attempt to draw with it.

(Note that all BViews attached to a window share the same connection to the Application Server. Calling **Flush()** or **Sync()** for any one of them flushes the buffer for all of them.)

See also: **BWindow::Flush()**, the **BBitmap** class

Invalidate()

```
void Invalidate(BRect rect)
void Invalidate(void)
```

Invalidates the *rect* portion of the view, causing update messages—and consequently **Draw()** notifications—to be generated for the BView and all descendants that lie wholly or partially within the rectangle. The rectangle is stated in the BView's coordinate system.

If no rectangle is specified, the BView's entire bounds rectangle is invalidated.

Since only BViews that are attached to a window can draw, only attached BViews can be invalidated.

See also: **Draw()**, **GetClippingRegion()**, **BWindow::UpdateIfNeeded()**

InvertRect()

```
void InvertRect(BRect rect)
```

Inverts all the colors displayed within the *rect* rectangle. A subsequent **InvertRect()** call on the same rectangle restores the original colors. This operation can be used to "highlight" a selection made as the user drags the mouse.

The rectangle is stated in the BView's coordinate system.

See also: **BScreen::ColorMap**

SetViewBitmap() , ClearViewBitmap()

```
void SetViewBitmap(const BBitmap *bitmap,
    BRect source,
    BRect destination,
    uint32 follow = B_FOLLOW_TOP | B_FOLLOW_LEFT,
    uint32 options = B_TILE_BITMAP)
void SetViewBitmap(const BBitmap *bitmap,
    uint32 follow = B_FOLLOW_TOP | B_FOLLOW_LEFT,
    uint32 options = B_TILE_BITMAP)
void ClearViewBitmap(void)
```

`SetViewBitmap()` sets the background bitmap for the view. The view bitmap is a background image for the view; all drawing in the view occurs over this bitmap. The background color is used to fill in the visible regions not covered by the background bitmap.

The background bitmap is passed in *bitmap*. The caller can delete *bitmap* after the function returns.

If a *source* rectangle is given, only that part of the bitmap is used. Otherwise, the entire bitmap is used as the background bitmap. The *destination* bitmap, if given, specifies the placement of the bitmap in the view. It need not be the same size as the *source* rectangle; scaling is performed automatically by the application server. If no *destination* is given, the image will be placed, unscaled, at the upper left corner of the view.

follow determines the behavior of *destination* as the view is resized; see the BView constructor for specifics. *options* specifies additional view options. Currently, only one option, **`B_TILE_BITMAP`**, is defined. If set, the view bitmap is tiled across the view.

`ClearViewBitmap()` clears the background bitmap for the view.

See also: "The Background Bitmap" in the "Drawing" section of this chapter, **`SetViewColor()`**

SetViewOverlay() , ClearViewOverlay()

```
void SetViewOverlay(const BBitmap *overlay,
    BRect source,
    BRect destination,
    rgb_color *colorKey,
    uint32 follow = B_FOLLOW_TOP | B_FOLLOW_LEFT,
    uint32 options = 0)
void SetViewOverlay(const BBitmap *bitmap,
    rgb_color *colorKey,
    uint32 follow = B_FOLLOW_TOP | B_FOLLOW_LEFT,
    uint32 options = 0)
void ClearViewOverlay(void)
```

`SetViewOverlay()` sets the overlay bitmap for the view. The overlay bitmap is superimposed on top of the view's contents. The *colorKey* is used to determine which color in the overlay should

be treated as transparent, allowing the view's contents to be visible.

The overlay bitmap is passed in *bitmap*. The caller can delete *bitmap* after the function returns.

If a *source* rectangle is given, only that part of the bitmap is used. Otherwise, the entire bitmap is used as the overlay bitmap. The *destination* bitmap, if given, specifies the placement of the overlay bitmap in the view. It need not be the same size as the *source* rectangle; scaling is performed automatically by the application server. If no *destination* is given, the image will be placed, unscaled, at the upper left corner of the view.

follow determines the behavior of *destination* as the view is resized; see the BView constructor for specifics. *options* specifies additional view options. The same options allowed by `SetViewBitmap()` are allowed here.

You can't use both a background bitmap and an overlay in the same view.

`ClearViewOverlay()` clears the overlay bitmap for the view.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BView Drawing-Related Functions Function Summary

Function Summary

Function	Description
AddLine()	Adds a line to the line array
AppendToPicture()	Begins recording a picture, appending to an existing BPicture
BeginLineArray()	Begins a line array
BeginPicture()	Begins recording a new picture
ClearViewBitmap()	Clears the background view bitmap
ClearViewOverlay()	Clears the background view overlay
CopyBits()	Copies image from one part of view to another
DrawBitmapAsync()	Draws a BBitmap in the view (asynchronous)
DrawBitmap()	Draws a BBitmap in the view (synchronous)
DrawChar()	Draws a character in the view
DrawPictureAsync()	Renders a BPicture in the view (asynchronous)
DrawPicture()	Renders a BPicture in the view (synchronous)
DrawString()	Draws a string in the view
EndLineArray()	Ends the line array
EndPicture()	Ends recording a picture
FillArc()	Fills in part of an ellipse
FillBezier()	Fills a region defined by a third degree

	Bezier curve
FillEllipse()	Fills an ellipse
FillPolygon()	Fills a polygon
FillRect()	Fills a rectangle
FillRegion()	Fills a region with a given pattern
FillRoundRect()	Fills a rounded rectangle
FillShape()	Fills a shape
FillTriangle()	Fills a triangle
Flush()	Flushes drawing queue, returning immediately
Invalidate()	Invalidates a portion of the view
InvertRect()	Inverts the colors of a rectangle in the view
SetViewBitmap()	Sets the background view bitmap
SetViewOverlay()	Sets the background view overlay
StrokeArc()	Outlines part of an ellipse
StrokeBezier()	Draws a third degree Bezier curve
StrokeEllipse()	Outlines an ellipse
StrokeLine()	Draws a line
StrokePolygon()	Draws a line along the edges of a polygon
StrokeRect()	Draws a line along the edges of a rectangle
StrokeRoundRect()	Draws a line along a rounded rectangle
StrokeShape()	Draws a line along the edges of a shape
StrokeTriangle()	Draws a line along the edges of a

	triangle
Sync()	Flushes drawing queue, returning when operations have completed

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Graphics State Functions

These functions alter or retrieve information on the graphics state (the set of internal variables controlling aspects of drawing) of a BView instance. To make it easier to find the functions of interest, the graphics state functions have been divided into the following subcategories:

- General and Miscellaneous
- Pen Functions
- Color
- Drawing Modes
- Text Functions
- Clipping Functions

General and Miscellaneous

- LineCapMode() see SetLineMode()
- LineJoinMode() see SetLineMode()
- LineMiterLimit() see SetLineMode()

- PushState() , PopState()

void **PushState**(void)
void **PopState**(void)

Saves and restores the state from the state stack. A state consists of the following: local and global origins, local and global scales, drawing mode, line cap and join modes, miter limit, pen size and location, foreground and background color, stipple pattern, local and global clipping regions, and the font context. When a state is saved to the stack, a new state context is created, with a local scale of zero, a local origin at (0,0), and no clipping region.

If the BView isn't attached to a window, these functions will crash the application.

- SetLineMode() , LineJoinMode() , LineCapMode() , LineMiterLimit()

void **SetLineMode**(cap_mode *lineCap*, join_mode *lineJoin*, float *miterLimit* = B_DEFAULT_MITER_LIMIT)
cap_mode **LineCapMode**(void) const
join_mode **LineJoinMode**(void) const
float **LineMiterLimit**(void) const

These methods implement support for PostScript-style line cap and join modes. The cap mode determines the shape of the endpoints of stroked paths, while the join mode determines the shape of the corners of the paths (i.e. where two lines meet).

The following values of cap_mode are defined:

B_ROUND_CAP	A semicircle is drawn around the endpoint. Its diameter is equal to the width of the line.
B_BUTT_CAP	The line is squared off and does not extend beyond the endpoint.
B_SQUARE_CAP	The line is squared off, extending past the endpoint for a distance equal to half the width of the line.

Additionally, the following values of join_mode are defined:

B_ROUND_JOIN	Acts identically to **B_ROUND_CAP**, except applied to joins.
B_MITER_JOIN	The lines are extended until they touch. If they meet at an angle greater than 2*arcsin(1/*miterLimit*), a bevel join is used instead.
B_BEVEL_JOIN	Butt end caps are used at the common endpoint and the empty area between the caps is filled with a triangle.
B_BUTT_JOIN	Acts identically to **B_BUTT_CAP**, except applied to joins.
B_SQUARE_JOIN	Acts identically to **B_SQUARE_CAP**, except applied to joins.

`SetLineMode()` sets the line and join modes and the miter limit while `LineCapMode()`, `LineJoinMode()`, and `LineMiterLimit()` return them. The line mode affects all of the Stroke... methods except for Arc, Ellipse, and RoundRect.

SetScale()

void **SetScale(**float *ratio***)**

Scales the coordinate system the view uses for drawing. The default scale is 1.0; smaller *ratio* values reduce the size of the drawing coordinate system; larger numbers magnify the system. For example, a ratio of 0.5 makes a subsequent drawing twice as small and moves the drawing closer to the origin, and 2.0 makes it twice as big and moves it away, as shown below.

The scaling ratio only affects subsequent drawing operations! Changing the scale doesn't affect the graphics already displayed in the view, the view's frame rectangle and clipping region, the placement and size of subviews, translation of mouse coordinates to and from view space, and so forth.

Multiple **SetScale()** calls don't compound within the same graphics state, but they do compound across pushed states:

```
aview->SetScale(0.5);
aview->SetScale(0.5);
/* aview's scaling is 0.5. */

bview->SetScale(0.5);
bview->PushState();
bview->SetScale(0.5);
/* view's scaling is 0.25. */
```

The BView must be attached to a window before you call this function.

Pen Functions

MovePenBy() , MovePenTo() , PenLocation()

void **MovePenBy(**float *horizontal*, float *vertical***)**
void **MovePenTo(**BPoint *point***)** void **MovePenTo(**float *x*, float *y***)**
BPoint **PenLocation(**void**)** const

These functions move the pen (without drawing a line) and report the current pen location.

MovePenBy() moves the pen *horizontal* coordinate units to the right and *vertical* units downward. If *horizontal* or *vertical* are negative, the pen moves in the opposite direction. **MovePenTo()** moves the pen to *point*—or to (*x*, *y*)—in the BView's coordinate system.

Some drawing functions also move the pen—to the end of whatever they draw. In particular, this is true of **StrokeLine()**, **DrawString()**, and **DrawChar()**. Functions that stroke a closed shape (such as **StrokeEllipse()**) don't move the pen.

The pen location is a parameter of the BView's graphics environment, which is maintained by both the Application Server and the BView. If the BView doesn't belong to a window, **MovePenTo()** and **MovePenBy()** cache the location, so that later, when the BView becomes attached to a window, it can be handed to the server to become the operable pen location for the BView. If the BView belongs to a window, these functions alter both the server parameter and the client-side cache.

PenLocation() returns the point where the pen is currently positioned in the BView's coordinate system. Because of the cache, this shouldn't entail contacting the server. The default pen position is (0.0, 0.0).

See also: **SetPenSize()**

SetPenSize() , PenSize()

virtual void **SetPenSize(**float *size***)**
float **PenSize(**void**)** const

SetPenSize() sets the size of the BView's pen—the graphics parameter that determines the thickness of stroked lines—and **PenSize()** returns the current pen size. The pen size is stated in coordinate units, but is translated to a device-specific number of pixels for each output device.

The pen tip can be thought of as a brush that's centered on the line path and held perpendicular to it. If the brush is broader than one pixel, it paints roughly the same number of pixels on both sides of the path.

The default pen size is 1.0 coordinate unit. It can be set to any nonnegative value, including 0.0. If set to 0.0, the size is translated to one pixel for all devices. This guarantees that it will always draw the thinnest possible line no matter what the resolution of the device.

Thus, lines drawn with pen sizes of 1.0 and 0.0 will look alike on the screen (one pixel thick), but the line drawn with a pen size of 1.0 will be 1/72 of an inch thick when printed, however many printer pixels that takes, while the line drawn with a 0.0 pen size will be just one pixel thick.

The pen size is a parameter of the BView's graphics environment maintained by the Application Server and cached by the BView. If the BView isn't attached to a window, `SetPenSize()` records the *size* so that later, when the BView is added to a window and becomes known to the server, the cached value can automatically be established as the operable pen size for the BView. If the BView belongs to a window, this function changes both the server and the cache.

See also: "The Pen" in the "Drawing" section of this chapter, `StrokeArc()`, `MovePenBy()`

Color

SetHighColor() , HighColor() , SetLowColor() , LowColor()

virtual void **SetHighColor(**rgb_color *color***)** inline void **SetHighColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 255**)**
rgb_color **HighColor(**void**)** const
virtual void **SetLowColor(**rgb_color *color***)** inline void **SetLowColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 255**)**
rgb_color **LowColor(**void**)** const

These functions set and return the current high and low colors of the BView. These colors combine to form a pattern that's passed as an argument to the `Stroke...()` and `Fill...()` drawing functions. The **`B_SOLID_HIGH`** pattern is the high color alone, and **`B_SOLID_LOW`** is the low color alone.

The default high color is black—*red*, *green*, and *blue* values all equal to 0. The default low color is white—*red*, *green*, and *blue* values all equal to 255.

The inline versions of **`SetHighColor()`** and **`SetLowColor()`** take separate arguments for the *red*, *blue*, and *green* color components; they work by creating an **`rgb_color`** data structure and passing it to the corresponding function that's declared `virtual`. Therefore, if you want to override either function, you should override the virtual version. (However, due to the peculiarities of C++, overriding any version of an overloaded function hides all versions of the function. For continued access to the nonvirtual version without explicitly specifying the "BView::" prefix, simply copy the inline code from **interface/View.h** into the derived class.)

The high and low colors are parameters of the BView's graphics environment, which is kept in the BView's shadow counterpart in the Application Server and cached in the BView. If the BView isn't attached to a window, **`SetHighColor()`** and **`SetLowColor()`** cache the *color* value so that later, when the BView is placed in a window and becomes known to the server, the cached value can automatically be registered as the current high or low color for the view. If the BView belongs to a window, these functions alter both the client-side and the server-side values.

`HighColor()` and **`LowColor()`** return the BView's current high and low colors. Because of the cache, this shouldn't entail contacting the Application Server.

See also: "Patterns" in the "Drawing" section of this chapter, **`SetViewColor()`**

SetViewColor() , ViewColor()

virtual void **SetViewColor(**rgb_color *color***)** inline void **SetViewColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 255**)**

rgb_color **ViewColor**(void) const

These functions set and return the view's background color. This is the color that's displayed when a view is erased during an update, or when the view is resized to expose new areas. The default view color is white (255,255,255). If you don't want the view to be erased in an update, set the view color to **B_TRANSPARENT_COLOR**. (Despite the name this doesn't actually make the view transparent.)

The inline version of **SetViewColor()** calls the virtual version. Thus, overriding the virtual version affects both versions. However, due to the peculiarities of C++, overriding any version of an overloaded function hides all versions of the function. To fix this, simply copy the inline code from **View.h** into your subclass.

ViewColor() returns the current background color.

See also: "The View Color" in the "Drawing" section of this chapter, **SetHighColor()**, **SetViewBitmap()**

Drawing Modes

DrawingMode() see SetDrawingMode()

SetBlendingMode() , GetBlendingMode()

virtual void **SetBlendingMode(**source_alpha *alphaSrcMode*, alpha_function *alphaFncMode***)**
virtual void **GetBlendingMode(**source_alpha **alphaSrcMode*, alpha_function **alphaFncMode***)

These two functions set and retrieve the graphics state variables which control the details of alpha transparency drawing. These variables will have an effect on drawing in the view only if the drawing mode has been set to **B_OP_ALPHA** by **SetDrawingMode()**.

alphaSrcMode is one of the following two constants, with associated meanings:

- **B_CONSTANT_ALPHA** : Use the alpha channel of the current high color as the transparency value for whatever is being drawn.
- **B_PIXEL_ALPHA** : When drawing a bitmap, use the alpha value associated with each pixel as the transparency value for that pixel. This can be used to obtain some interesting variable transparency effects.

alphaFncMode is one of the following two constants, with associated meanings:

- **B_ALPHA_OVERLAY** : The "normal" mode, used when drawing a transparent image or shape over an opaque background.
- **B_ALPHA_COMPOSITE** : Used when blending two or more transparent images together offscreen, to produce a new transparent image that will later be drawn onscreen using the **B_ALPHA_OVERLAY** setting.

You can find a full tutorial on alpha drawing at the top level of The Interface Kit documentation.

SetDrawingMode() , DrawingMode()

virtual void **SetDrawingMode(**drawing_mode *mode***)**
drawing_mode **DrawingMode(**void**)** const

These functions set and return the BView's drawing mode, which can be any of the following eleven constants:

B_OP_COPY	**B_OP_MIN**
B_OP_OVER	**B_OP_MAX**

B_OP_ERASE	B_OP_ADD
B_OP_INVERT	B_OP_SUBTRACT
B_OP_SELECT	B_OP_BLEND
B_OP_ALPHA	

The drawing mode is an element of the BView's graphics environment, which both the Application Server and the BView keep track of. If the BView isn't attached to a window, SetDrawingMode() caches the *mode*. When the BView is placed in a window and becomes known to the server, the cached value is automatically set as the current drawing mode. If the BView belongs to a window, SetDrawingMode() makes the change in both the server and the cache.

DrawingMode() returns the current mode. Because of the cache, this generally doesn't entail a trip to the server.

The default drawing mode is B_OP_COPY. It and the other modes are explained under "Drawing Modes" in the "Drawing" section of this chapter.

Text Functions

The functions below relate specifically to using text in BView objects.

ForceFontAliasing()

```
void ForceFontAliasing(bool enable)
```

ForceFontAliasing() is used in conjunction with printing. When called with a value of **true**, if causes subsequent printing to be done *without* antialiasing printed characters. This is normally what is desired with high-resolution printers, to guarantee that the edges of printed characters appear sharp. Calling ForceFontAliasing() with an argument of **false** turns antialiasing back on, which may be desirable with lower-resolution printers.

Note that ForceFontAliasing() does not affect characters or strings drawn to the screen.

See also: The BPrintJob class.

GetFont() see SetFont()

GetFontHeight()

```
void GetFontHeight(font_height *fontHeight) const
```

Gets the height of the BView's font. This function provides the same information as BFont's GetHeight(). The following code

```
font_height height;
myView->GetFontHeight(&height);
```

is equivalent to:

```
font_height height;
BFont font;
myView->GetFont(&font);
font.GetHeight(&height);
```

See the BFont class for more information.

See also: BFont::GetHeight()

GetStringWidths() see StringWidth()

SetFont() , GetFont()

virtual void **SetFont(**const BFont **font*, uint32 *properties* = B_FONT_ALL**)**

void **GetFont(**BFont **font***)**

SetFont() sets the BView's current font so that it matches the specified *properties* of the *font* BFont object. The *properties* mask is formed by combining the following constants:

B_FONT_FAMILY_AND_STYLE	**B_FONT_SPACING**
B_FONT_SIZE	**B_FONT_ENCODING**
B_FONT_SHEAR	**B_FONT_FACE**
B_FONT_ROTATION	**B_FONT_FLAGS**

Each constant corresponds to a settable property of the BFont object. The default mask, **B_FONT_ALL**, is a shorthand for all the properties (including any that might be added in future releases). If the mask is 0, **SetFont()** won't set the BView's font.

GetFont() copies the BView's current font to the BFont object passed as an argument. Modifying this copy doesn't modify the BView's font; it takes an explicit **SetFont()** call to affect the BView.

For example, this code changes the size of a BView's font and turns antialiasing off:

```
BFont font;
myView->GetFont(&font);
font.SetSize(67.0);
font.SetFlags(B_DISABLE_ANTIALIASING);
myView->SetFont(&font, B_FONT_SIZE | B_FONT_FLAGS);
```

Since the BFont object that this example code alters is a copy of the BView's current font, it's not strictly necessary to name the properties that are different when calling **SetFont()**. However, it's more efficient and better practice to do so.

The font is part of the BView's graphic environment. Like other elements in the environment, it can be set whether or not the BView is attached to the window. Graphics parameters are kept by the Application Server and also cached by the BView object.

See also: the BFont class, **get_font_family()**

SetFontSize()

void **SetFontSize(**float *points***)**

Sets the size of the BView's font to *points*. This function is a shorthand for a **SetFont()** call that just alters the font size. For example, this line of code

```
myView->SetFontSize(12.5);
```

does the same thing as:

```
BFont font;
font.SetSize(12.5);
myView->SetFont(&font, B_FONT_SIZE);
```

See also: the BFont class, **`SetFont()`**

StringWidth() , GetStringWidths()

float **StringWidth(**const char **string***)* const float **StringWidth(**const char **string*, int32 *length***)** const
void **GetStringWidths(**char **stringArray*[], int32 *lengthArray*[], int32 *numStrings*, float *widthArray*[]

These functions measure how much room is required to draw a *string*, or a group of strings, in the BView's current font. They're equivalent to the identically named set of functions defined in the BFont class, except that they assume the BView's font. For example, this line of code

```
float width;
width = myView->StringWidth("Be"B_UTF8_REGISTERED);
```

produces the same result as:

```
float width;
BFont font;
myView->GetFont(&font);
width = font.StringWidth("Be"B_UTF8_REGISTERED);
```

See the BFont class for details.

See also: **`BFont::StringWidth()`**, **`BFont::GetEscapements()`**

TruncateString()

void **TruncateString(**BString **inOutString*, uint32 *mode*, float *width***)** const

Truncates the BString *inOutString* to be no wider than *width* pixels. The *mode* flags control how the string is truncated.

`B_TRUNCATE_BEGINNING`	Cut from the beginning of the string until it fits within the specified width.
`B_TRUNCATE_MIDDLE`	Cut from the middle of the string.
`B_TRUNCATE_END`	Cut from the end of the string.

B_TRUNCATE_SMART	Cut anywhere, but do so intelligently, so that all the strings remain different after being cut. For example, if a set of similar path names are passed in the *inputStringArray*, this mode would attempt to cut from the identical parts of the path names and preserve the parts that are different. This mode also pays attention to word boundaries, separators, punctuation, and the like. However, it's not implemented for the current release.

Clipping Functions

ClipToPicture() , ClipToInversePicture()

void **ClipToPicture(**BPicture **picture*, BPoint *where* = B_ORIGIN, bool *sync* = true**)**

void **ClipToInversePicture(**BPicture **picture*, BPoint *where* = B_ORIGIN, bool *sync* = true**)**

Modifies the view's clipping region by intersecting the current clipping region with the pixels drawn by *picture* (in the case of **ClipToPicture**) or with everything outside the pixels drawn by *picture* (in the case of **ClipToInversePicture**), to produce the new clipping region.

Note that **BPicture** instances are, by their nature, resolution independent; when the **ClipToPicture** or **ClipToInversePicture** command is invoked, *picture* is effectively drawn at the same resolution as the invoking view, and the bitmap produced by that action is used to modify the clipping region. You may think of *picture* as executing its drawing instructions on a surface that starts out as completely transparent; at the end of the process, each pixel on the drawing surface will either be either completely transparent, or will be at least somewhat opaque. The pixels which are at least somewhat opaque are those which were "drawn" by *picture*.

If *sync* is false, the functions will execute asynchronously; normally they execute synchronously (i.e. wait for the drawing actions to be completed by the Application Server.)

See also: **BPicture**, **BeginPicture()**

ConstrainClippingRegion()

virtual void **ConstrainClippingRegion(**BRegion **region***)**

Restricts the drawing that the BView can do to *region*.

The Application Server keeps track of a clipping region for each BView that's attached to a window. It clips all drawing the BView does to that region; the BView can't draw outside of it.

By default, the clipping region contains only the visible area of the view and, during an update, only the area that actually needs to be drawn. By passing a *region* to this function, an application can further restrict the clipping region. When calculating the clipping region, the server intersects it with the *region* provided. The BView can draw only in areas common to the *region* passed and the clipping region as the server would otherwise calculate it. The region passed can't expand the clipping region beyond what it otherwise would be.

The clipping region is additionally affected by any items on the state stack. If any saved states contain clipping regions, then the actual clipping region used by the Application Server is the intersection of the local clipping region (as set by this method) and the regions stored on the state stack.

If called during an update, **ConstrainClippingRegion()** restricts the clipping region only for the duration of the update.

Calls to **ConstrainClippingRegion()** are not additive; each *region* that's passed replaces the one that was passed in the previous call. Passing a `NULL` pointer removes the previous region without replacing it. The function works only for BViews that are attached to a window.

See also: **GetClippingRegion()**, **Draw()**

GetClippingRegion()

void **GetClippingRegion(**BRegion ***region)** const

Modifies the BRegion object passed as an argument so that it describes the current local clipping region of the BView, the region where the BView is allowed to draw. It's most efficient to allocate temporary BRegions on the stack:

```
BRegion clipper;
GetClippingRegion(&clipper);
. . .
```

Ordinarily, the clipping region is the same as the visible region of the view, the part of the view currently visible on-screen. The visible region is equal to the view's bounds rectangle minus:

- The frame rectangles of its children,
- Any areas that are clipped because the view doesn't lie wholly within the frame rectangles of all its ancestors in the view hierarchy, and
- Any areas that are obscured by other windows or that lie in a part of the window that's off-screen.

The clipping region can be smaller than the visible region if the program restricted it by calling **ConstrainClippingRegion()**. It will exclude any area that doesn't intersect with the region passed to **ConstrainClippingRegion()**.

The clipping region is additionally modified by any items on the state stack. If any saved states contain clipping regions, then the actual clipping region used by the Application Server is the intersection of the local clipping region (as set by **ConstrainClippingRegion()**) and the regions stored on the state stack.

While the BView is being updated, the clipping region contains just those parts of the view that need to be redrawn. This may be smaller than the visible region, or the region restricted by **ConstrainClippingRegion()**, if:

- The update occurs during scrolling. The clipping region will exclude any of the view's visible contents that the Application Server is able to shift to their new location and redraw automatically.

- The view rectangle has grown (because, for example, the user resized the window larger) and the update is needed only to draw the new parts of the view.
- The update was caused by `Invalidate()` and the rectangle passed to `Invalidate()` didn't cover all of the visible region.
- The update was necessary because `CopyBits()` couldn't fill all of a destination rectangle.

If, while updating is ongoing, you call the view's parent's `GetClippingRegion()` function, the resulting region indicates only the area of that view that requries updating, and so forth. In other words, this change in behavior (from returning the true clipping region to returning the update region) is recursive up the view hierarchy.

This function works only if the BView is attached to a window. Unattached BViews can't draw and therefore have no clipping region.

See also: `ConstrainClippingRegion()`, `Draw()`, `Invalidate()`

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Variables, Constants, and Defined Types

This section lists the various global variables, constants, and types that the Interface Kit defines.

Global Variables

System Fonts

Declared in: be/interface/Font.h

const BFont ***be_plain_font**
const BFont ***be_bold_font**
const BFont ***be_fixed_font**

These global BFont objects are created when the BApplication object is constructed. They encapsulate the three system fonts—the plain font which is used for labels and small stretches of text in the user interface, the bold font which is used for window and group titles, and the fixed font which is used in Terminal windows and other places where a fixed-width font is required.

These objects cannot be modified directly, nor are they modified when the user redefines a system font. The user's changed preferences don't affect running applications.

Constants

alert_type (BAlert)

alignment Constants

Declared in: be/interface/InterfaceDefs.h

Constant
B_ALIGN_LEFT
B_ALIGN_RIGHT
B_ALIGN_CENTER

These constants define the **alignment**alert_type (BAlert) data type. They determine how lines of text and labels are aligned by BTextView, BStringView, and BMenuField objects.

See also: BTextView::SetAlignment()

button_width Constants

Declared in: be/interface/InterfaceDefs.h

Constant
B_WIDTH_AS_USUAL
B_WIDTH_FROM_LABEL
B_WIDTH_FROM_WIDEST

These constants define the button_width type. They determine how the width of a set of buttons will be set—whether they're set to a minimal width, a width just sufficient to accommodate the button's own label, or a width sufficient to accommodate the widest label of all the buttons.

The constants are used by the BAlert and BTabView classes.

border_style Constants

Declared in: be/interface/InterfaceDefs.h

Constant
B_PLAIN_BORDER
B_FANCY_BORDER
B_NO_BORDER

These constants define how BBox objects and BScrollViews are bordered.

See also: the BScrollView and BBox classes

Character Constants

Declared in: be/interface/InterfaceDefs.h

Constant	Character value
B_BACKSPACE	0x08 (same as 'b')
B_ENTER	0x0a (same as 'n')
B_RETURN	0x0a (synonym for B_ENTER)
B_SPACE	0x20 (same as ' ')
B_TAB	0x09 (same as 't')
B_ESCAPE	0x1b
B_LEFT_ARROW	0x1c
B_RIGHT_ARROW	0x1d
B_UP_ARROW	0x1e
B_DOWN_ARROW	0x1f
B_INSERT	0x05
B_DELETE	0x7f
B_HOME	0x01
B_END	0x04
B_PAGE_UP	0x0b
B_PAGE_DOWN	0x0c
B_FUNCTION_KEY	0x10
B_UTF8_ELLIPSIS	"xE2x80xA6"
B_UTF8_OPEN_QUOTE	"xE2x80x9C"
B_UTF8_CLOSE_QUOTE	"xE2x80x9D"
B_UTF8_COPYRIGHT	"xC2xA9"
B_UTF8_REGISTERED	"xC2xAE"
B_UTF8_TRADEMARK	"xE2x84xA2"
B_UTF8_SMILING_FACE	"xE2x98xBB"
B_UTF8_HIROSHI	"xE5xBCx98"

Constants in the first group stand for the ASCII characters they name. They're defined only for characters that normally don't have visible symbols.

Constants in the second group are Unicode UTF-8 encodings of common characters that have multibyte representations. These constants are strings that can be concatenated with other strings—for example, to set a button label that ends in an ellipsis:

```
myButton->SetLabel("Options"B_UTF_ELLIPSIS);
```

See also: "Function Key Constants" below

---

color_space

Declared in: be/interface/GraphicsDefs.h

Constant	Description
B_NO_COLOR_SPACE	Undefined color space
B_RGB32	xRGB 8:8:8:8, stored as little endian uint32
B_RGBA32	ARGB 8:8:8:8, stored as little endian uint32
B_RGB24	Currently unused
B_RGB16	xRGB 5:6:5, stored as little endian uint16
B_RGB15	RGB 1:5:5:5, stored as little endian uint16
B_RGBA15	ARGB 1:5:5:5, stored as little endian uint16
B_CMAP8	256-color index into the color table.
B_GRAY8	256-color greyscale value.
B_GRAY1	Each bit represents a single pixel, where 0 is the view's low color and 1 is th color.
B_RGB32_BIG	xRGB 8:8:8:8, stored as big endian uint32
B_RGBA32_BIG	ARGB 8:8:8:8, stored as big endian uint32
B_RGB24_BIG	Currently unused
B_RGB16_BIG	RGB 5:6:5, stored as big endian uint16
B_RGB15_BIG	xRGB 1:5:5:5, stored as big endian uint16
B_RGBA15_BIG	ARGB 1:5:5:5, stored as big endian uint16
B_YCbCr422	Currently unused.
B_YCbCr411	Currently unused.
B_YCbCr444	Currently unused.
B_YCbCr420	Currently unused.
B_YUV422	Currently unused.
B_YUV411	Currently unused.
B_YUV420	Currently unused.
B_YUV444	Currently unused.
B_YUV9	Currently unused.
B_YUV12	Currently unused.
B_UVL24	Currently unused.
B_UVL32	Currently unused.
B_UVLA32	Currently unused.
B_LAB24	Currently unused.
B_LAB32	Currently unused.
B_LABA32	Currently unused.

B_HSI24	Hue, saturation, and intensity: x:8:8:8 format
B_HSI32	Hue, saturation, and intensity: 8:8:8 format
B_HSIA32	Hue, saturation, intensity, and alpha: 8:8:8:8 format
B_HSV24	Hue, saturation, and value: x:8:8:8 format
B_HSV32	Hue, saturation, and value: 8:8:8 format
B_HSVA32	Hue, saturation, value, and alpha: 8:8:8:8 format
B_HLS24	Hue, lightness, and saturation: x:8:8:8 format
B_HLS32	Hue, lightness, and saturation: 8:8:8 format
B_HLSA32	Hue, lightness, saturation, and alpha: 8:8:8:8 format
B_CMY24	Cyan, magenta, and yellow: x:8:8:8 format
B_CMY32	Cyan, magenta, and yellow: 8:8:8 format
B_CMYA32	Cyan, magenta, yellow, and alpha: 8:8:8:8 format
B_CMYK32	Cyan, magenta, yellow, and black: 8:8:8:8 format
B_MONOCHROME_1_BIT	For compatibility only: use **B_GRAY1**
B_GRAYSCALE_8_BIT	For compatibility only; use **B_GRAY8**
B_COLOR_8_BIT	For compatibility only; use **B_CMAP8**
B_RGB_32_BIT	For compatibility only; use **B_RGB32**
B_RGB_16_BIT	For compatibility only; use **B_RGB15**
B_BIG_RGB_32_BIT	For compatibility only; use **B_RGB32_BIG**
B_BIG_RGB_16_BIT	For compatibility only; use **B_RGB15_BIG**

These constants define the **color_space** data type. A color space describes three properties of screens and bitmap images:

- How many bits of information there are per pixel (the depth of the image)
- How those bits are to be interpreted (whether as colors or on a grayscale, what the color components are, and so on).
- How are components are arranged

A number of these aren't actually usable as screen display color spaces, but are intended for use in identifying the color format for video media buffers and other offscreen graphics data.

See the "Color Spaces" section in the "Drawing" section of this chapter for a fuller explanation of these color spaces.

See also: "Colors" in the "Drawing" section of this chapter, the BBitmap and BScreen classes

---

Control Values

Declared in: be/interface/Control.h

Constant	Value
B_CONTROL_ON	1

B_CONTROL_OFF	0

These constants define the bipolar states of a typical control device.

See also: `BControl::SetValue()`

Cursor Transit Constants

Declared in: be/interface/View.h

Constant	Meaning
B_ENTERED_VIEW	The cursor has just entered a view.
B_INSIDE_VIEW	The cursor has moved within the view.
B_EXITED_VIEW	The cursor has left the view

These constants describe the cursor's transit through a view. Each `MouseMoved()` notification includes one of these constants as an argument, to inform the BView whether the cursor has entered the view, moved while inside the view, or exited the view.

See also: **`BView::MouseMoved()`**

Dead-Key Mapping

Declared in: be/interface/InterfaceDefs.h

Constant
B_CONTROL_TABLE
B_OPTION_CAPS_SHIFT_TABLE
B_OPTION_CAPS_TABLE
B_OPTION_SHIFT_TABLE
B_OPTION_TABLE
B_CAPS_SHIFT_TABLE
B_CAPS_TABLE
B_SHIFT_TABLE
B_NORMAL_TABLE

These constants determine which combinations of modifiers can cause a key to be the "dead" member of a two-key combination.

See also: `get_key_map()`

---

drawing_mode Constants

Declared in: be/interface/GraphicsDefs.h

Constant	Constant
B_OP_COPY	B_OP_ADD
B_OP_OVER	B_OP_SUBTRACT
B_OP_ERASE	B_OP_BLEND
B_OP_INVERT	B_OP_MIN
B_OP_SELECT	B_OP_MAX
B_OP_ALPHA	

These constants define the **drawing_mode** data type. The drawing mode is a BView graphics parameter that determines how the image being drawn interacts with the image already in place in the area where it's drawn. The various modes are explained under "Drawing Modes" in the "Drawing" section of this chapter.

See also: "Drawing Modes" in the "Drawing" section of this chapter, **BView::SetDrawingMode()**

---

font_direction Constants

Declared in: be/interface/Font.h

Constant
B_FONT_LEFT_TO_RIGHT
B_FONT_RIGHT_TO_LEFT

These constants tell whether a font is used for text that's read left-to-right or right-to-left. Thus is an inherent property of the font.

See also: **BFont::Direction()**

Font Encodings

Declared in: be/interface/Font.h

Constant
B_UNICODE_UTF8
B_ISO_8859_1
B_ISO_8859_2
B_ISO_8859_3
B_ISO_8859_4
B_ISO_8859_5
B_ISO_8859_6
B_ISO_8859_7
B_ISO_8859_8
B_ISO_8859_9
B_ISO_8859_10

B_MACINTOSH_ROMAN

The constants name the various character encodings that the BeOS supports. B_UNICODE_UTF8 is the default encoding. It matches ASCII values for 7-bit character codes but uses multiple bytes to encode other values in the Unicode standard.

See also: BFont::SetEncoding(), "Character Encoding" in the "Responding to the User" section of this chapter

---

Font Flags

Declared in: be/interface/Font.h

Constant
B_DISABLE_ANTIALIASING
B_IS_FIXED
B_HAS_TUNED_FONT

The first flag, B_DISABLE_ANTIALIASING, is passed to a BFont object to turn antialiasing off. Antialiasing should be turned off when printing, but should generally be left on when drawing to the screen.

The other two flags enable get_font_family() and get_font_style() to give information about a font. B_IS_FIXED indicates that the font is nonproportional. B_HAS_TUNED_FONT indicates that the family or style has one or more tuned fonts—bitmap fonts that have been adjusted to look good on the screen—for some set of font properties (such as size and shear).

See also: BFont::SetFlags(), get_font_family()

Font Name Lengths

Declared in: be/interface/Font.h

Constant	Value
B_FONT_FAMILY_NAME_LENGTH	63
B_FONT_STYLE_NAME_LENGTH	63

These constants define the maximum length of names for font families and styles, exclusive of a null terminator. They're used in the definition of the font_family and font_style types.

See also: font_family under "Defined Types" below

Font Properties

Declared in: be/interface/View.h

Constant

B_FONT_FAMILY_AND_STYLE
B_FONT_SIZE
B_FONT_SHEAR
B_FONT_ROTATION
B_FONT_SPACING
B_FONT_ENCODING
B_FONT_FACE
B_FONT_FLAGS
B_FONT_ALL

These constants list the font properites that can be set for a BView individually or in combination. The constants form a mask that's passed, along with a BFont object, to BView's **SetFont()** and BTextView's **SetFontAndColor()** functions. For example:

```
myView->SetFont(theFont, B_FONT_SIZE | B_FONT_ENCODING);
```

B_FONT_ALL stands for all properties of the BFont.

See also: **BView::SetFont()**

Font Spacing Modes

Declared in: be/interface/Font.h

Constant
B_CHAR_SPACING
B_STRING_SPACING
B_BITMAP_SPACING
B_FIXED_SPACING

These constants enumerate the four modes for positioning characters in a line of text.

See also: **BFont::SetSpacing()**

Function Key Constants

Declared in: be/interface/InterfaceDefs.h

Constant	Constant
B_F1_KEY	**B_F9_KEY**
B_F2_KEY	**B_F10_KEY**
B_F3_KEY	**B_F11_KEY**
B_F4_KEY	**B_F12_KEY**
B_F5_KEY	**B_PRINT_KEY** (the "Print Screen" key)

B_F6_KEY	**B_SCROLL_KEY** (the "Scroll Lock" key)
B_F7_KEY	**B_PAUSE_KEY**
B_F8_KEY	

These constants stand for the various keys that are mapped to the **B_FUNCTION_KEY** character. When the **B_FUNCTION_KEY** character is reported in a key-down event, the application can determine which key produced the character by testing the key code against these constants. (Control-*p* also produces the **B_FUNCTION_KEY** character.)

See also: "Character Mapping" in the **Keyboard Information** appendix

---

Interface Messages

Declared in: be/app/AppDefs.h

Constant	Constant
B_ZOOM	**B_KEY_DOWN**
B_MINIMIZE	**B_KEY_UP**
B_WINDOW_RESIZED	**B_MOUSE_DOWN**
B_WINDOW_MOVED	**B_MOUSE_UP**
B_WINDOW_ACTIVATED	**B_MOUSE_MOVED**
B_QUIT_REQUESTED	**B_VIEW_RESIZED**
B_SCREEN_CHANGED	**B_VIEW_MOVED**
B_WORKSPACE_ACTIVATED	**B_VALUE_CHANGED**
B_WORKSPACES_CHANGED	**B_PULSE**

These constants identify interface messages—system messages that are delivered to BWindow objects. Each constant conveys an instruction to do something in particular (`B_ZOOM`) or names a type of event (`B_KEY_DOWN`).

See also: "Interface Messages" in the "Responding to the User" section of this chapter

---

list_view_type Constants

Declared in: be/interface/ListView.h

Constant
B_SINGLE_SELECTION_LIST
B_MULTIPLE_SELECTION_LIST

These constants distinguish between lists that permit the user to select only one item at a time and those that allow multiple items to be selected.

See also: the BListView class

Main Screen

Declared in: be/interface/Screen.h

const screen_id **B_MAIN_SCREEN_ID**

This constant stands for the main screen, the screen that has the origin of the screen coordinate system at its left top corner. (Currently only one screen can be attached to the computer and it is the main screen.)

See also: `screen_id`

---

menu_bar_border Constants

Declared in: be/interface/MenuBar.h

Constant	Meaning
B_BORDER_FRAME	Put a border inside the entire frame rectangle.
B_BORDER_CONTENTS	Put a border around the group of items only.
B_BORDER_EACH_ITEM	Put a border around each item.

These constants can be passed as an argument to BMenuBar's `SetBorder()` function.

See also: `BMenuBar::SetBorder()`

---

menu_layout Constants

Declared in: be/interface/Menu.h

Constant	Meaning
B_ITEMS_IN_ROW	Menu items are arranged horizontally, in a row.
B_ITEMS_IN_COLUMN	Menu items are arranged vertically, in a column.
B_ITEMS_IN_MATRIX	Menu items are arranged in a custom fashion.

These constants define the `menu_layout` data type. They distinguish the ways that items can be arranged in a menu or menu bar—they can be laid out from end to end in a row like a typical menu bar, stacked from top to bottom in a column like a typical menu, or arranged in some custom fashion like a matrix.

See also: the BMenu and BMenuBar constructors

Modifier States

Declared in: be/interface/InterfaceDef.h

Constant	Constant
B_SHIFT_KEY	**B_OPTION_KEY**

B_LEFT_SHIFT_KEY	B_LEFT_OPTION_KEY
B_RIGHT_SHIFT_KEY	B_RIGHT_OPTION_KEY
B_CONTROL_KEY	B_COMMAND_KEY
B_LEFT_CONTROL_KEY	B_LEFT_COMMAND_KEY
B_RIGHT_CONTROL_KEY	B_RIGHT_COMMAND_KEY
B_CAPS_LOCK	B_MENU_KEY
B_SCROLL_LOCK	
B_NUM_LOCK	

These constants designate the Shift, Option, Control, Command, and Menu modifier keys and the lock states set by the Caps Lock, Scroll Lock, and Num Lock keys. They're typically used to form a mask that describes the current, or required, modifier states.

For each variety of modifier key, there are constants that distinguish between the keys that appear at the left and right of the keyboard, as well as one that lumps both together. For example, if the user is holding the left Control key down, both `B_CONTROL_KEY` and `B_LEFT_CONTROL_KEY` will be set in the mask.

See also: `modifiers()`, `BWindow::AddShortcut()`, the BMenu constructor

Mouse Buttons

Declared in: be/interface/View.h

Constant
B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

These constants name the mouse buttons. Buttons are identified, not by their physical positions on the mouse, but by their roles in the user interface.

See also: `BView::GetMouse()`, `set_mouse_map()`

---

orientation Constants

Declared in: be/interface/InterfaceDef.h

Constant
B_HORIZONTAL
B_VERTICAL

These constants define the `orientation` data type that distinguishes between the vertical and horizontal orientation of graphic objects. It's currently used only to differentiate scroll bars.

See also: the BScrollBar and BScrollView classes

Pattern Constants

Declared in: be/interface/GraphicsDef.h

const pattern **B_SOLID_HIGH** = { 0xff, 0xff, 0xff, 0xff, 0xff,0xff, 0xff, 0xff }
const pattern **B_SOLID_LOW** = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
const pattern **B_MIXED_COLORS** = { 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 }

These constants name the three standard patterns defined in the Interface Kit.

`B_SOLID_HIGH` is a pattern that consists of the high color only. It's the default pattern for all BView drawing functions that stroke lines and fill shapes.

`B_SOLID_LOW` is a pattern with only the low color. It's used mainly to erase images (to replace them with the background color).

`B_MIXED_COLORS` alternates pixels between the high and low colors in a checkerboard pattern. The result is a halftone midway between the two colors. This pattern can produce fine gradations of color, especially when the high and low colors are set to two colors that are already quite similar.

See also: "Patterns" in the "Drawing" section of this chapter

Resizing Modes

Declared in: be/interface/View.h

Constants
B_FOLLOW_LEFT
B_FOLLOW_RIGHT
B_FOLLOW_LEFT_RIGHT
B_FOLLOW_H_CENTER
B_FOLLOW_TOP
B_FOLLOW_BOTTOM
B_FOLLOW_TOP_BOTTOM
B_FOLLOW_V_CENTER
B_FOLLOW_ALL
B_FOLLOW_NONE

These constants are used to set the behavior of a view when its parent is resized. They're explained under the BView constructor.

See also: the BView constructor, **`BView::SetResizingMode()`**

Screen Spaces

Declared in: be/interface/GraphicsDef.h

B_8_BIT_640x400			
B_8_BIT_640x480	B_15_BIT_640x480	B_16_BIT_640x480	B_32_BIT_640x480
B_8_BIT_800x600	B_15_BIT_800x600	B_16_BIT_800x600	B_32_BIT_800x600
B_8_BIT_1024x768	B_15_BIT_1024x768	B_16_BIT_1024x768	B_32_BIT_1024x768
B_8_BIT_1152x900	B_15_BIT_1152x900	B_16_BIT_1152x900	B_32_BIT_1152x900
B_8_BIT_1280x1024	B_15_BIT_1280x1024	B_16_BIT_1280x1024	B_32_BIT_1280x1024
B_8_BIT_1600x1200	B_15_BIT_1600x1200	B_16_BIT_1600x1200	B_32_BIT_1600x1200

These constants are currently used to configure the screen—to set its depth and the size of the pixel grid it displays—as well as to report which configurations are possible. However, they may not be supported in the future. 15-bit depths are not currently supported

See also: `set_screen_space()`, `get_screen_info()`

Scroll Bar Constants

Declared in: be/interface/ScrollBar.h

Constant
B_H_SCROLL_BAR_HEIGHT
B_V_SCROLL_BAR_WIDTH

These constants record the recommended thickness of scroll bars. They should be used to help define the frame rectangles passed to the BScrollBar constructor.

See also: the BScrollBar class

String Truncation Constants

Declared in: be/interface/Font.h

Constant
B_TRUNCATE_END
B_TRUNCATE_BEGINNING
B_TRUNCATE_MIDDLE
B_TRUNCATE_SMART

These constants instruct a BFont where it should remove characters from a set of strings to shorten them.

See also: `BFont::GetTruncatedStrings()`

---

tint_color() constants

Declared in: be/interface/InterfaceDefs.h

Constant	Meaning
B_LIGHTEN_MAX_TINT	The maximum amount of tinting lighter. A color component value of 216 will become 255.
B_LIGHTEN_2_TINT	The second largest amount of lightening. A color component value of 216 will become 240.
B_LIGHTEN_1_TINT	A small amount of lightening. A color component value of 216 will become 232.
B_NO_TINT	Has no effect on the color.
B_DARKEN_1_TINT	The least amount of darkening. A color component value of 184 will become zero.
B_DARKEN_2_TINT	Darker. A color component value of 216 will become 152.
B_DARKEN_3_TINT	Even darker. A color component value of 216 will become 128.
B_DARKEN_4_TINT	Darker still A color component value of 216 will become 96.
B_DARKEN_MAX_TINT	Wow, that's really dark. A color component value of 216 will become zero.
B_DISABLED_LABEL_TINT	Use this constant when generating colors with which to disable text labels.
B_HIGHLIGHT_BACKGROUND_TINT	Use this constant to create a dark color for use as a background color on selected data.
B_DISABLED_MARK_TINT	Use this to create a color for a disabled checkmark or radio button mark (or other tick-mark type items).

These constants are used when calling **`tint_color()`** to alter an existing color, especially for creating disabled objects or highlighting backgrounds.

See also: **`tint_color()`**

Transparency Constants

Declared in: be/interface/GraphicsDefs.h

const uint8 **B_TRANSPARENT_8_BIT**
const rgb_color **B_TRANSPARENT_32_BIT**

These constants set transparent pixel values in a bitmap image. `B_TRANSPARENT_8_BIT` designates a transparent pixel in the **`B_CMAP8`** color space, and `B_TRANSPARENT_32_BIT` designates a transparent pixel in the **`B_RGB32`** color space.

Transparency is explained in the "Drawing Modes" part of the "Drawing" section of this chapter. Drawing modes other than **`B_OP_COPY`** preserve the destination image where a source

bitmap is transparent.

See also: "Drawing Modes" in the "Drawing" section of this chapter, the BBitmap class, **`BView::SetViewColor()`**

---

color_which constants

Declared in: be/interface/InterfaceDefs.h

Constant	Meaning
`B_PANEL_BACKGROUND_COLOR`	The background color used for view panels.
`B_MENU_BACKGROUND_COLOR`	The background color used for menus.
`B_MENU_SELECTION_BACKGROUND_COLOR`	The color used for highlighting menu items when selected.
`B_MENU_ITEM_TEXT_COLOR`	The color used for drawing menu item labels when not selected.
`B_MENU_SELECTED_ITEM_TEXT_COLOR`	The color used for drawing menu item labels when selected.
`B_WINDOW_TAB_COLOR`	The color used to draw the window's title tab.
`B_KEYBOARD_NAVIGATION_COLOR`	The color used to draw the mark indicating which control is focused for keyboard navigation.
`B_DESKTOP_COLOR`	The current desktop color.

When calling **`ui_color()`** to obtain the color for a user interface element, these constants are used to select the desired element.

See also: **`ui_color()`**

View Flags

Declared in: be/interface/View.h

Constant	Meaning
`B_FULL_UPDATE_ON_RESIZE`	Include the entire view in the clipping region.
`B_WILL_DRAW`	Allow the BView to draw.
`B_PULSE_NEEDED`	Report pulse events to the BView.
`B_FRAME_EVENTS`	Report view-resized and view-moved events.
`B_NAVIGABLE`	Let users navigate to the view with the Tab key.
`B_NAVIGABLE_JUMP`	Mark the view for Control-Tab navigation.

These constants can be combined to form a mask that sets the behavior of a BView object. They're explained in more detail under the class constructor. The mask is passed to the constructor, or to the `SetFlags()` function.

See also: the BView constructor, **`BView::SetFlags()`**

Window Flags

Declared in: be/interface/Window.h

Constant	Constant
B_NOT_MOVABLE	B_NOT_CLOSABLE
B_NOT_H_RESIZABLE	B_NOT_ZOOMABLE
B_NOT_V_RESIZABLE	B_NOT_MINIMIZABLE
B_NOT_RESIZABLE	B_WILL_FLOAT
B_WILL_ACCEPT_FIRST_CLICK	

These constants set the behavior of a window. They can be combined to form a mask that's passed to the BWindow constructor.

See also: the BWindow constructor

window_type Constants

Declared in: be/interface/Window.h

Constant	Meaning
B_MODAL_WINDOW	The window is a modal window.
B_BORDERED_WINDOW	The window has a border but no title tab.
B_TITLED_WINDOW	The window has a border and a title tab.
B_DOCUMENT_WINDOW	The window has a border, tab and resize knob.

These constants describe the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor

Workspace Constants

Declared in: be/interface/Window.h

Constant
B_CURRENT_WORKSPACE
B_ALL_WORKSPACES

These constants are used—along with designations of specific workspaces—to associate a set of one or more workspaces with a BWindow.

See also: the BWindow constructor, `BWindow::SetWorkspaces()`

Defined Types

alert_type

Declared in: be/interface/Alert.h

```
typedef enum {. . .} alert_type
```

These constants name the various types of alert panel.

See also: "`alert_type` Constants", the BAlert constructor

alignment

Declared in: be/interface/InterfaceDefs.h

```
typedef enum {. . .} alignment
```

Alignment constants determine where lines of text are placed in a view.

See also: "`alignment` Constants", **`BTextView::SetAlignment()`**

button_width

Declared in: be/interface/Alert.h

```
typedef enum {. . .} button_width
```

These constants name the methods that can be used to determine how wide to make the buttons in an alert panel.

See also: "`button_width` Constants", the BAlert constructor

color_map

Declared in: be/interface/GraphicsDefs.h

```
typedef struct {
        int32 id;
        rgb_color color_list[256];
        uint8 inversion_map[256];
        uint8 index_map[32768];
    } color_map
```

This structure contains information about the 8-bit color context for a particular screen. All applications that display on the screen share the same color map.

See also: `BScreen::ColorMap()`

---

color_space

Declared in: be/interface/GraphicsDefs.h

```
typedef enum {. . .} color_space
```

Color space constants determine the depth and interpretation of bitmap images. They're described under "Colors" in the "Drawing" section of this chapter.

See also: "`color_space` Constants"

---

drawing_mode

Declared in: be/interface/GraphicsDefs.h

```
typedef enum {. . .} drawing_mode
```

The drawing mode determines how source and destination images interact.

See also: "Drawing Modes" in the "Drawing" section of this chapter, "`drawing_mode` Constants"

---

edge_info

Declared in: be/interface/Font.h

```
typedef struct {
        float left;
        float right;
    } edge_info
```

This structure records information about the location of a character outline within the horizontal space allotted to the character. Edges separate one character from adjacent characters on the left and right. They're explained under the `GetEdges()` function in the BFont class.

See also: `BFont::GetEscapements()`

---

escapement_delta

Declared in: be/interface/Font.h

```
typedef struct {
        float nonspace;
        float space;
    } escapement_delta
```

This structure contains values that should be added to the width of each character in a string when the string is drawn.

See also: `BView::DrawString()`, `BFont::GetEscapements()`

---

font_direction

Declared in: be/interface/Font.h

```
typedef enum {...} font_direction
```

This type distinguishes between fonts that are read left-to-right (`B_FONT_LEFT_TO_RIGHT`) and those that are read right-to-left (`B_FONT_RIGHT_TO_LEFT`).

See also: `BFont::Direction()`

---

font_family

Declared in: be/interface/InterfaceDefs.h

```
typedef char font_family[B_FONT_FAMILY_LENGTH + 1]
```

This type defines a string long enough to hold the name of a font family—64 characters including the null terminator.

See also: `get_font_family()`

---

font_height

Declared in: be/interface/Font.h

```
typedef struct {
        float ascent;
        float descent;
        float leading;
    } font_height
```

This type combines the three vertical measurements that determine the height of a line of text.

See also: `BFont::GetHeight()`

---

font_style

Declared in: be/interface/InterfaceDefs.h

```
typedef char font_style[B_FONT_STYLE_LENGTH + 1]
```

This type defines a string long enough to hold the name of a font style—64 characters including the null terminator.

See also: `get_font_style()`

---

key_info

Declared in: be/interface/View.h

```
typedef struct {
        uint32 modifiers;
        uint8 key_states[16];
    } key_info
```

This structure is used to get information about the current state of the keyboard in the absence of `B_KEY_DOWN` messages.

See also: `get_key_info()`

---

key_map

Declared in: be/interface/InterfaceDefs.h

```
typedef struct {
        uint32 version;
        uint32 caps_key;
        uint32 scroll_key;
        uint32 num_key;
        uint32 left_shift_key;
        uint32 right_shift_key;
        uint32 left_command_key;
        uint32 right_command_key;
        uint32 left_control_key;
        uint32 right_control_key;
        uint32 left_option_key;
        uint32 right_option_key;
        uint32 menu_key;
        uint32 lock_settings;
        int32 control_map[128];
        int32 option_caps_shift_map[128];
        int32 option_caps_map[128];
        int32 option_shift_map[128];
        int32 option_map[128];
        int32 caps_shift_map[128];
        int32 caps_map[128];
        int32 shift_map[128];
        int32 normal_map[128];
        int32 acute_dead_key[32];
```

```
    int32 grave_dead_key[32];
    int32 circumflex_dead_key[32];
    int32 dieresis_dead_key[32];
    int32 tilde_dead_key[32];
    uint32 acute_tables;
    uint32 grave_tables;
    uint32 circumflex_tables;
    uint32 dieresis_tables;
    uint32 tilde_tables;
} key_map
```

This structure maps the physical keys on the keyboard to their functions in the user interface. It holds the tables that assign characters to key codes, set up dead keys, and determine which keys function as modifiers. There's just one key map shared by all applications running on the same machine.

See also: `get_key_map()`

---

menu_bar_border

Declared in: be/interface/MenuBar.h

```
typedef enum {. . .} menu_bar_border
```

This type enumerates the ways that a menu bar can be bordered.

See also: `BMenuBar::SetBorder()`, "`menu_bar_border` Constants" above

---

menu_info

Declared in: be/interface/Menu.h

```
typedef struct {
    float font_size;
    font_name font;
    rgb_color background_color;
    int32 separator;
    bool click_to_open;
    bool triggers_always_shown;
} menu_info
```

This structure records the user's menu preferences.

See also: `set_menu_info()` , the BMenu class

---

menu_layout

Declared in: be/interface/Menu.h

```
typedef enum {. . .} menu_layout
```

This type distinguishes the various ways that items can arranged in a menu or menu bar.

See also: the BMenu class, "`menu_layout` Constants" above

---

mouse_map

Declared in: be/interface/InterfaceDefs.h

```
typedef struct {
        uint32 left;
        uint32 right;
        uint32 middle;
    } mouse_map
```

This structure maps mouse buttons to their roles as the `B_PRIMARY_MOUSE_BUTTON`, `B_SECONDARY_MOUSE_BUTTON`, or `B_TERTIARY_MOUSE_BUTTON`.

See also: `set_mouse_map()`

---

orientation

Declared in: be/interface/InterfaceDefs.h

```
typedef enum {. . .} orientation
```

This type distinguishes between the `B_VERTICAL` and `B_HORIZONTAL` orientation of scroll bars.

See also: the BScrollBar and BScrollView classes

---

pattern

Declared in: be/interface/GraphicsDefs.h

```
typedef struct {
        uchar data[8];
    } pattern
```

A pattern is a arrangement of two colors—the high color and the low color—in an 8-pixel by 8-pixel square. Pixels are specified in rows, with one byte per row and one bit per pixel. Bits marked 1 designate the high color; those marked 0 designate the low color. An example and an illustration are given under "Patterns" in the "Drawing section of this chapter.

See also: "Pattern Constants" above

---

print_file_header

Declared in: be/interface/PrintJob.h

```
typedef struct {
        long version;
        long page_count;
        long _reserved_1_;
        long _reserved_2_;
        long _reserved_3_;
        long _reserved_4_;
        long _reserved_5_;
    } print_file_header
```

This structure defines the header information for a print job. Although declared publicly, it currently is used only internally by the BPrintJob class.

---

rgb_color

Declared in: be/interface/GraphicsDefs.h

```
typedef struct {
        uint8 red;
        uint8 green;
        uint8 blue;
        uint8 alpha;
    } rgb_color
```

This type specifies a full 32-bit color. Each component can have a value ranging from a minimum of 0 to a maximum of 255.

The `alpha` component, which is designed to specify the coverage of the color (how transparent or opaque it is), is currently ignored. However, an `rgb_color` can be made completely transparent by assigning it the special value, **`B_TRANSPARENT_32_BIT`**.

See also: **`BView::SetHighColor()`**

---

screen_id

Declared in: be/interface/Screen.h

```
typedef struct {
        int32 id;
    } screen_id
```

This type is a unique identifier for a screen. The constant **`B_MAIN_SCREEN_ID`** is a **`screen_id`** that identifies the main screen.

See also: the BScreen class

---

screen_info

Declared in: be/interface/InterfaceDefs.h

```
typedef struct {
        color_space mode;
        BRect frame;
        uint32 spaces;
        float min_refresh_rate;
        float max_refresh_rate;
        float refresh_rate;
        uchar h_position;
        uchar v_position;
        uchar h_size;
        uchar v_size;
    } screen_info
```

This structure holds information about a screen. Its fields are explained under the `get_screen_info()` global function.

See also: `get_screen_info()`

---

scroll_bar_info

Declared in: be/interface/InterfaceDefs.h

```
typedef struct {
        bool proportional;
        bool double_arrows;
        int32 knob;
        int32 min_knob_size;
    } scroll_bar_info
```

This structure captures the user's preferences for how scroll bars should behave and appear.

See also: `set_scroll_bar_info()`, the BScrollBar class

---

text_run

Declared in: be/interface/TextView.h

```
typedef struct {
        int32 offset;
        BFont font;
        rgb_color color;
    } text_run
```

The BTextView class uses this structure to keep track of a sequence of characters that are displayed in the same font and color.

See also: `BTextView::SetRunArray()`

---

text_run_array

Declared in: be/interface/TextView.h

```
typedef struct {
        int32 count;
        text_run runs[];
    } text_run_array
```

The BTextView class uses this structure to associate character formats with the text it displays.

See also: `BTextView::SetRunArray()`

---

tuned_font_info

Declared in: be/interface/Font.h

```
typedef struct {
        float size;
        float shear;
        float rotation;
        int32 flags;
        int16 face;
    } tuned_font_info
```

This structure lists the properties of a font that has a counterpart that's been tuned to improve its appearance on-screen. The tuned counterpart will be used whenever the font is the BView's current font and `DrawString()` draws to the screen.

See also: `BFont::GetTunedInfo()`, `get_font_family()`

---

window_type

Declared in: be/interface/Window.h

```
typedef enum {. . .} window_type
```

This type describes the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor, "`window_type` Constants"

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTextView

Derived from: BView > BHandler
Declared in: be/interface/TextView.h
Library: libbe.so
Summary: *more...*

A BTextView object displays text on-screen, and provides these text manipulating features:

- It lets the user enter, select, and edit text from the keyboard and mouse.
- It supports standard Cut, Copy, Paste, Delete, and Select All editing commands
- It provides an Undo mechanism.

By default, a BTextView displays all its text in a single font and color. The `SetStylable()` turns on support for multiple character formats.

Paragraph formats—such as alignment and tab widths—are uniform for all text the BTextView displays. These properties can be set, but the setting always applies to the entire text.

The BTextView class isn't multi-thread safe; don't issue BTextView calls on a BTextView object from multiple threads, or you may see unusual behavior; in general, only the thread that created the BTextView should issue calls on it.

Offsets

The BTextView locates particular characters in its text buffer by offsets from the beginning of the data. The offsets count bytes, not characters, and begin at 0. A single character is identified by the offset of the first byte of the character. A group of characters—the current selection, for example—is delimited by the offsets that bound its first and last characters; all characters beginning with the first offset up to, but not including, the last offset are part of the group.

For example, suppose the BTextView contains the following text in Unicode UTF-8 encoding,

- The BeOS(TM) is . . .

and "BeOS(TM) " is selected. `GetSelection()` would return 4 and 11 as the offsets that enclose the selection. The character 'B' occupies the fourth byte of text and the space following the trademark symbol is the eleventh byte of text. The characters in "BeOS" are each encoded in one byte, but '(TM) ' takes up three bytes in UTF-8. Thus the five-character selection occupies 7 bytes (and offsets) of text.

Although offsets count bytes, they can also be thought of as designating positions between characters. The position at the beginning of the text is offset 0, the position between the

space and the 'B' in the example above is at offset 4, the position between the '(TM) ' and the space is at offset 11, and so on. Thus, even if no characters are selected, the insertion point (and location of the caret) can still be designated by an offset.

Most BTextView functions expect the offsets they're passed to mark positions between characters; the results are not defined if a character-internal offset is specified instead.

Graphics Primitives

The BTextView's mechanism for formatting and drawing text uses the graphics primitives it inherits from the BView class. However, it largely presents its own API for determining the appearance of the text it draws. You should not attempt to affect the BTextView by calling primitive BView functions like **`MovePen()`**, **`SetFont()`**, or **`SetHighColor()`**. Instead, use BTextView functions like **`SetFontAndColor()`** and let the object take care of formatting and drawing the text.

The one inherited function that can influence the BTextView is **`SetViewColor()`**. This function determines the background against which the text is drawn and the color that is used in antialiasing calculations.

Resizing

A BTextView can be made to resize itself to exactly fit the text that the user enters. This is sometimes appropriate for small one-line text fields. See the **`MakeResizable()`** function.

BTextView and BScrollBars

If you add scrollbars to control the position of the BTextView's document—this includes using a BTextView as the target of a BScrollView—the BTextView will update the scrollbars for you. (Note that it doesn't *own* the scrollbars; you still have to delete them yourself if you created them.) When the BTextView is first displayed and thereafter resized, the scrollbars' ranges, step sizes, and scroller positions and proportions are automatically reset to reflect the BTextView object's bounds. Attempts to set these parameters directly (through **`BScrollBar::SetRange()`** etc.), are *worse* than ignored; they're actually applied, and then (at some point) the BTextView will notice the change in the scrollbars and reset them. Looks like flicker to me.

Shortcuts and Menu Items

When it's the focus view for its window, a BTextView automatically responds to a set of keyboard shortcuts:

- Command+*x* cuts text and copies it to the clipboard
- Command+*c* copies text to the clipboard without cutting it
- Command+*v* pastes text taken from the clipboard
- Command+*a* selects all of the text in the BTextView
- Command+z undoes the previous action

If you create menu items for these actions, you have to assign the BMenuItem shortcuts and command constants yourself:

Action	Constant	Shortcut
Cut	`B_CUT`	Command+*x*
Copy	`B_COPY`	Command+*c*
Paste	`B_PASTE`	Command+*v*
Select all	`B_SELECT_ALL`	Command+*a*
Undo	`B_UNDO`	Command+z

Hook Functions

AcceptsDrop()

Determines whether a BMessage that's dragged to the BTextView has data that the object can insert; can be reimplemented to prevent the BTextView from accepting dropped text or to allow it to accept data it currently doesn't understand.

AcceptsPaste()

Determines whether the BTextView can take data from the clipboard; can be reimplemented to prevent text from being pasted or to make the BTextView accept data it currently doesn't understand.

CanEndLine()

Determines where a line can end; can be overridden to follow a different criterion than the default.

DeleteText()

Deletes characters from the text; can be augmented to preview the deletions and accept or reject them before the text is removed from the display. This function is called to carry out all deletions.

FindWord()

Finds the boundaries of a word when the user double-clicks; can be overridden to redefine what a word is or to provide a definition of a word for other languages.

GetDragParameters()

Invoked when a drag is initiated. You can override the function to supply parameters, such as a bitmap, that are used during the drag.

InsertText()

Inserts new characters into the text; can be augmented to preview the characters the user types, pastes, or drops and accept or reject them before they're added to the display. All insertions pass through this function.

Undo()

Invoked when the object gets a **B_UNDO** message. You can augment or replace the default undo response by implementing this function.

Constructor and Destructor

BTextView()

```
BTextView(BRect frame, const char *name, BRect textRect,
    uint32 resizingMode, uint32 flags)
BTextView(BRect frame, const char *name, BRect textRect,
    const BFont *font, const rgb_color *color,
    uint32 resizingMode, uint32 flags)
BTextView(BMessage *archive)
```

Initializes the BTextView to the *frame* rectangle, stated in its eventual parent's coordinate system, assigns it an identifying *name*, sets its resizing behavior to *resizingMode* and its drawing behavior with *flags*. These four arguments—*frame*, *name*, *resizingMode*, and *flags*—are identical to those declared for the BView class and are passed to the BView constructor. The *frame*, *name*, and *resizingMode* arguments are passed to the BView class unchanged, but two flags are added to the *flags* argument—`B_FRAME_EVENTS`, so that the BTextView can reformat the text when it's resized, and `B_PULSE_NEEDED`, so that the caret marking the insertion point can "blink" in time with `B_PULSE` messages. Later, `AttachedToWindow()` will set the window's pulse rate to 500,000 microseconds.

The text rectangle, *textRect*, is stated in the BTextView's coordinate system. It determines where text in placed within the view's bounds rectangle:

- The first line of text is placed at the top of the text rectangle. As additional lines of text are entered into the view, the text grows downward and may actually extend beyond the bottom of the rectangle.
- The left and right sides of the text rectangle determine where lines of text are placed within the view. Lines can be aligned to either side of the rectangle, or they can be centered between the two sides. See the `SetAlignment()` function.
- When lines wrap on word boundaries, the width of the text rectangle determines the maximum length of a line; each line of text can be as long as the rectangle is wide. When word wrapping isn't turned on, lines can extend beyond the boundaries of the text rectangle. See the `SetWordWrap()` function.

The bottom of the text rectangle is ignored; it doesn't limit the amount of text the view can contain. The text can be limited by the number of characters, but not by the number of lines.

If a default *font* is provided, the BTextView will display its text in that font, unless another font is later set. Similarly, if a default *color* is specified, the text will be displayed in that color, unless the color is subsequently changed. If the *font* is `NULL` or not specified, the BTextView uses the system plain font, `be_plain_font`. If the *color* pointer is `NULL` or not specified, the text is drawn in black.

The constructor establishes the following default properties for a new BTextView:

- The text is selectable and editable. (See `MakeSelectable()` and `MakeEditable()`.)
- Multiple character formats are not permitted. (See `SetStylable()`.)
- The text is left-aligned. (See `SetAlignment()`.)
- The tab width is 28.0 coordinate units. (See `SetTabWidth()`.)
- Word wrapping is turned on. (See `SetWordWrap()`.)
- Automatic indenting is turned off. (See `SetAutoindent()`.)
- The maximum amount of data is permitted. (See `SetMaxBytes()`.)
- The view doesn't grow to accommodate more characters. (See `MakeResizable()`.)
- All characters the user may type are acceptable. (See `DisallowChar()`.)

See also: `AttachedToWindow()`, `SetFontAndColor()`, the BView constructor

~BTextView()

virtual **~BTextView()**

Frees the memory the BTextView allocated to hold the text and to store information about it.

Static Functions

FlattenRunArray() , UnflattenRunArray()

```
static void *FlattenRunArray(const text_run_array *runs, int32 *numBytes = NULL)
static text_run_array *UnflattenRunArray(const void *data, int32 *numBytes = NULL)
```

These functions flatten and unflatten a `text_run_array` structure so that it can be treated as an untyped stream of bytes. A `text_run_array` that's saved on-disk will be valid when the user reboots the machine only if it's saved as flat data. Both functions return pointers to memory they allocate (with `malloc()`). The caller is responsible for freeing the memory when it's no longer needed.

`FlattenRunArray()` flattens the *runs* `text_run_array` and returns the flat data. `UnflattenRunArray()` reconstructs a `text_run_array` from previously flattened *data* and returns a pointer to the structure.

If a *numBytes* argument is provided, both functions place the number of bytes they allocated for the data in the variable that *numBytes* refers to.

See also: `SetRunArray()`

Instantiate()

```
static BArchivable *Instantiate(BMessage *archive)
```

Returns a new BTextView object, allocated by `new` and created by the version of the constructor that takes a BMessage *archive*. However, if the archive doesn't contain data for a BTextView object, the return value will be `NULL`.

See also: `BArchivable::Instantiate()`, `Archive()`

UnflattenRunArray() see FlattenRunArray()

Member Functions

AcceptsDrop() , AcceptsPaste()

virtual bool **AcceptsDrop(**const BMessage *_message_**)**

virtual bool **AcceptsPaste(**BClipboard *_clipboard_**)**

These functions look at the data in their arguments, and return **true** if its "acceptable." The default implementations return `true` if both of the following are true:

- the BTextView is editable
- the *message* or *clipboard* contains **B_MIME_TYPE** data stored under the name "text/plain".

If the data isn't acceptable, the drop or paste operation is aborted.

`AcceptsDrop()` is invoked in two places: When a message is dragged over the BTextView, and when the message is dropped on the object. **AcceptsPaste()** is called when a **B_PASTE** message is received. If you augment these functions to accept more types, you'll also have to augment the **MessageReceived()** and **Paste()** functions to insert the text found in *message* or *clipboard*.

Alignment() see SetAlignment()

AllowChar() see DisallowChar()

Archive()

virtual status_t **Archive(**BMessage *_archive_, bool _deep_ = true**)** const

Calls the inherited version of **Archive()** and stores the BTextView in the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Completes the initialization of the BTextView object after it becomes attached to a window. This function sets up the object so that it can correctly format text and display it. Among other things, it sets the drawing mode to **B_OP_COPY**. If the BTextView is targeted by scroll bars, it adjusts them so that they're accurately set up for scrolling the text.

Because the BTextView uses pulses to animate (or "blink") the caret, the vertical line that marks the current insertion point, this function also enables pulsing in the window and fixes

the pulse rate at 2 per second.

AttachedToWindow() is called for you when the BTextView becomes part a window's view hierarchy; you shouldn't call it yourself, though you can override it. A function that's implemented by a derived class should begin by incorporating the BTextView version:

```
void MyText::AttachedToWindow()
{
   BTextView::AttachedToWindow()
   . . .
}
```

If it doesn't, the BTextView won't be able to properly display the text.

See also: **`BView::AttachedToWindow()`**, **`SetFontName()`**

ByteAt() see Text()

CanEndLine()

virtual bool **CanEndLine(**int32 *offset***)**

Returns `true` if the character at *offset* can be the last character in a line of text, and `false` if not. Sometimes this depends on whether the next character (if there is one) can begin a line. This function is called as the BTextView figures out where to break lines, but only if word wrapping is turned on.

As implemented, **CanEndLine()** allows the following ASCII characters to end a line regardless of the context:

B_SPACE	=	<	/	&
B_TAB	+	>		*
B_ENTER	–	^	|	'0'

The default implementation also understands the line-ending conventions for Chinese and Japanese. Because these languages are written without the spaces that typically end lines in other languages, lines can potentially break anywhere. However, certain characters are prohibited from ending a line and others are prohibited from beginning a new line. **CanEndLine()** prevents lines from ending either on a prohibited ending character or on the character before a prohibited beginning character.

Derived classes can override this function to apply different criteria for where lines end, possibly looking at the context of the *offset* character. You can also augment the current implementation so that it understands the conventions for other languages.

If you override this function to look to the left or right of the character at *offset*, be sure to check that you haven't stepped beyond the range of the text. For example, this version of the

function makes sure that the first hyphen of a pair doesn't break a line:

```
bool MyTextView::CanEndLine(int32 offset)
{
   if ( ByteAt[offset] == '-' ) {
      if ( TextLength() - offset > 1 ) {
            if ( ByteAt[offset + 1] == '-' )
               return false;
      }
   }
   return (baseClass::CanEndLine(offset));
}
```

See also: `SetWordWrap()`

- Clear() see Cut()
- ColorSpace() see SetColorSpace()
- Copy() see Cut()
- CountLines() see GoToLine()
- CurrentLine() see GoToLine()

- Cut() , Copy() , Paste() , Clear()

virtual void **Cut(**BClipboard *_clipboard_**)**
virtual void **Copy(**BClipboard *_clipboard_**)**
virtual void **Paste(**BClipboard *_clipboard_**)**
virtual void **Clear(**void**)**

These functions implement the standard cut, copy, paste, and clear commands. `Cut()` and `Copy()` both copy the current selection to the specified *clipboard*; `Cut()` also deletes the text from the BTextView and removes it from the display. The text is entered in the clipboard as `B_MIME_TYPE` data under the name "text/plain". `Paste()` looks in the *clipboard* for just this type of data and pastes it into the text—but only if `AcceptsPaste()` returns `true`. The new text replaces the current selection, or is placed at the site of the current insertion point.

If the BTextView supports multiple character formats, `Cut()` and `Copy()` also place a `text_run_array` structure describing the formats of the copied text in the clipboard—as `B_MIME_TYPE` data under the name "application/x-vnd.Be-text_run_array". If the BTextView that takes text from the clipboard supports multiple formats, `Paste()` looks for the `text_run_array` in the clipboard and sets the formats of the pasted text accordingly.

In most cases, the *clipboard* argument will be identical to the global `be_clipboard` object.

The `Clear()` function simply removes the currently-selected text from the BTextView without affecting any clipboard. If there's no selection, nothing happens.

See also: **`AcceptsPaste()`**, "Shortcuts and Menu Items" in the overview

---

- Delete() see Insert()
- DeleteText() see InsertText()

- DetachedFromWindow()

virtual void **`DetachedFromWindow(`**void**`)`**

Resets the cursor to the standard hand image (**`B_HAND_CURSOR`**) if it's above the BTextView when the BTextView is removed from the window.

See also: **`BView::DetachedFromWindow()`**

- DisallowChar() , AllowChar()

void **`DisallowChar(`**uint32 *aChar*`)`
void **`AllowChar(`**uint32 *aChar*`)`

These functions inform the BTextView whether the user should be allowed to enter *aChar* into the text. By default, all characters are allowed. Call **`DisallowChar()`** for each character you want to prevent the BTextView from accepting, preferably when first setting up the object. Although declared as `uint32`, *aChar* must be a character encoded in a single byte; it can't be a 16-bit Unicode value or a multibyte UTF-8 string.

`AllowChar()` reverses the effect of **`DisallowChar()`**.

Alternatively, and for more control over the context in which characters are accepted or rejected, you can implement an **`InsertText()`** function for the BTextView. **`InsertText()`** is called for all insertions, including each character the user types, all text the user drags to the BTextView, and all attempts to paste from the clipboard.

See also: **`AcceptsChar()`**

- DoesAutoindent() see SetAutoindent()
- DoesUndo() see SetDoesUndo()
- DoesWordWrap() see SetWordWrap()

- Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the BTextView.

See also: **`BView::Draw()`**

FindWord()

virtual void **FindWord(**int32 *offset*, int32 **start*, int32 **finish***)**

Looks for a sequence of characters that qualifies as a word—that is, a sequence that the user can double-click to select—that includes the character at *offset*. This function places the offset of the word's first character in the variable that *start* refers to and the offset following the last character in the word in the variable that *finish* refers to. If the *offset* character can't be part of a word, the *start* and *finish* offsets will be identical.

As implemented, this function allows the user to select a group of similar characters with a double-click. For example, in the following line of malformed text,

```
"You what!!?"
```

it would allow the user to select the words "You" and "what," the group of spaces between the words, and the group of punctuation marks at the end.

The function also defines similar groups of Japanese characters that can be selected together.

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Overrides the BView version of this function to reset the ranges of the BTextView's scroll bars and to update the sizes of their proportional knobs whenever the size of the BTextView changes.

See also: **`BView::FrameResized()`**

GetDragParameters()

protected: virtual void **GetDragParameters(**BMessage **drag*,
BBitmap ***bitmap*,
BPoint **point*,
BHandler ***handler***)**

`GetDragParameters()` is called when a drag session is initiated. The values that it retrieves (by reference in the arguments) are passed on to **`BView::DragMessage()`**. If you don't supply a

bitmap (if you set **bitmap* to **NULL**), the outline rectangle version of DragMessage() is used.

GetSelection()

void **GetSelection**(int32 **start*, int32 **finish*)

Provides the current selection by writing the offset before the first selected character into the variable referred to by *start* and the offset after the last selected character into the variable referred to by *finish*. If no characters are selected, both offsets will record the position of the current insertion point.

If the text isn't selectable, both offsets will be 0.

See also: Select()

Get see Text()

GetTextRegion()

void **GetTextRegion**(int32 *start*, int32 *finish*, BRegion **region*) const

Calculates the region where the run of characters beginning at the *start* offset and ending at the *finish* offset would be displayed within the BTextView's coordinate system, and modifies the BRegion object passed as the third argument, *region*, so that it represents that region.

See also: TextHeight()

GoToLine() , CountLines() , CurrentLine()

void **GoToLine**(int32 *index*)
int32 **CurrentLine**(void) const
int32 **CountLines**(void) const

GoToLine() moves the insertion point to the beginning of the line at *index*. The first line has an index of 0, the second line an index of 1, and so on. If the *index* is out-of-range, the insertion point is moved to the beginning of the line with the nearest in-range index—that is, to either the first or the last line.

CurrentLine() returns the index of the line where the first character of the selection—or the character following the insertion point—is currently located.

CountLines() returns how many lines of text the BTextView currently contains.

Like other functions that change the selection, GoToLine() doesn't automatically scroll the display to make the new selection visible. Call ScrollToSelection() to be sure that the user can

see the start of the selection.

See also: `ScrollToSelection()`

HideTyping() , IsTypingHidden()

void **HideTyping(**bool *flag***)**
bool **IsTypingHidden(**void**)** const

The first of these functions sets whether the user can see the text in the BTextView; the second returns whether or not the text is currently visible. When *flag*'s state is **true**, text contained by the BTextView isn't visible.

Highlight()

inline void **Highlight(**int32 *start*, int32 *finish***)**

Highlights (or unhighlights) the characters between the *start* and *finish* offsets. This is the function that the BTextView calls to highlight and unhighlight the current selection. You don't need to call it yourself for this purpose. It's in the public API just in case you may need to highlight a range of text in some other circumstance.

If the text is not currently highlighted, this function highlights it. But if the text is already highlighted, it unhighlights it. If you highlight some text, be sure to unhighlight it before the next editorial change; the BTextView will not do it for you.

See also: `Select()`, **TextRegion()**

Insert() , Delete()

void **Insert(**const char **text*, const text_run_array **runs* = NULL**)** void **Insert(**const char **text*, int32 *length*, const text_run_array **runs* = NULL**)** void **Insert(**int32 *offset*, const char **text*, int32 *length*, const text_run_array **runs* = NULL**)**
void **Delete(**void**)** void **Delete(**int32 *start*, int32 *finish***)**

`Insert()` adds *length* bytes of *text* to the BTextView—or if a *length* isn't specified, all the characters of the *text* string up to the null character that terminates it. The text is inserted at *offset*—or at the beginning of the current selection if an *offset* isn't specified. The current selection is not deleted and the insertion is not selected.

The inserted characters are displayed in the fonts and colors specified in the accompanying *runs* array, provided the BTextView allows multiple character formats. If multiple formats aren't allowed, the *runs* array is ignored. If multiple formats are allowed but a *runs* array isn't provided, the insertion is displayed in the font and color in force at the point of insertion. This

generally means the font and color of the first character of the selection, or of the character immediately preceding the *offset* character.

Offsets in the *runs* array should describe the text being inserted; in other words, the first offset should be 0. See SetRunArray() for a description of the text_run_array structure.

Insert() doesn't assume responsibility for the *text* data or the *runs* array. It copies the information it needs.

Delete() removes the characters bounded by the *start* and *finish* offsets from the display and deletes them from the BTextView's text, without copying them to the clipboard. If the *start* and *finish* offsets are the same, nothing is deleted. If offsets are not provided, Delete() deletes the current selection.

See also: SetText(), Cut(), SetRunArray()

InsertText() , DeleteText()

protected: virtual void **InsertText(**const char **text*,
int32 *length*,
int32 *offset*,
const text_run_array **runs***)**

protected: virtual void **DeleteText(**int32 *start*, int32 *finish***)**

These protected functions are the vehicles through which the BTextView performs every insertion and deletion of text (with one exception). You can augment them in a subclass to take note of pending changes to the text, and to modify or prevent the change.

Don't call Insert(), Delete() or any other high-level text-manipulating function in your implementation.

InsertText() adds *length* bytes of *text* to the BTextView, inserting it at *offset* within the text buffer. The font and color of the inserted characters may be described by an accompanying *runs* array. If the BTextView doesn't support multiple character formats, the *runs* array is ignored. If multiple formats are supported but the *runs* array is NULL, the text is displayed in the font and color of the character preceding *offset* (or of the first character, if *offset* is 0.)

The offsets in the *runs* data structure are relative to the inserted text; that is, the first offset in the array is 0, not *offset*.

InsertText() is called for every insertion, except one. The exception occurs when SetText() takes text from a file; in this case the text goes directly from the file to the BTextView; it's not stored in a temporary buffer while InsertText() is called.

DeleteText() removes the text bounded by the *start* and *finish* offsets. It fails if the offsets don't differ, or if the *finish* offset isn't greater than the *start* offset. This function is called for every deletion.

See also: Insert(), Delete()

IsEditable() see MakeEditable()

IsSelectable() see MakeSelectable()

KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes***)**

Enters text at the current selection in response to the user's typing. This function is called from the window's message loop for every report of a key-down event—typically once for every character the user types. However, it does nothing unless the BTextView is the focus view and the text it contains is editable.

If the character encoded in the *bytes* string is an editing instruction, `KeyDown()` takes the appropriate action:

- If the character is from one of the arrow keys (`B_UP_ARROW`, `B_LEFT_ARROW`, `B_DOWN_ARROW`, or `B_RIGHT_ARROW`), it extends the selection or moves the insertion point in the appropriate direction, depending on the modifiers
- If the character is `B_BACKSPACE` or `B_DELETE`, it deletes the current selection—or the character preceding or following the current insertion point.
- If the character comes from one of the paging keys (`B_HOME`, `B_END`, `B_PAGE_UP`, or `B_PAGE_DOWN`), it scrolls the display.

Otherwise, it checks whether the character was registered as unacceptable (by `DisallowChar()`). If not disallowed, it calls the `InsertText()` hook function to enter the character into the text and display it. Derived classes can preview about-to-be-inserted characters by overriding `InsertText()`.

See also: `BView::KeyDown()`, `InsertText()`, `DisallowChar()`

LineAt() , PointAt() , OffsetAt()

int32 **LineAt(**int32 *offset***)** const
int32 **LIneAt(**BPoint *point***)** const
BPoint **PointAt(**int32 *offset*, float **height* = NULL**)** const
int32 **OffsetAt(**BPoint *point***)** const int32 **OffsetAt(**int32 *index***)** const

These functions translate between coordinate values, text offsets, and line indices. `LineAt()` returns the index of the line containing the character at *offset* in the text, or the line located at the specified *point* in the BTextView's coordinate system. Line indices begin at 0.

PointAt() returns the coordinate location of the character at *offset*. The point is the left top corner of a rectangle enclosing the character and is stated in the BTextView's coordinate system. The *x*-coordinate of the point is the position on the baseline where the character is (or would be) drawn; its *y*-coordinate is the top of the line where the *offset* character is located. If a *height* argument is provided, PointAt() returns the height of the line by reference.

OffsetAt() returns the offset to the character that begins the *index* line, or to the character displayed at *point*.

LineHeight() , TextHeight()

float **LineHeight(**int32 *index* = 0**)** const
float **TextHeight(**int32 *firstIndex*, int32 *lastIndex***)** const

LineHeight() returns the height of the line of text at *index*, or the first line if an index isn't specified. Line indices begin at 0. The height is stated in coordinate units and depends on the font. It's the sum of how far characters can ascend above and descend below the baseline, plus the amount of leading that separates lines. If more than one font is used on the line, the ascent is taken from the tallest font and the descent and leading from the deepest.

TextHeight() returns the height of the set of lines from *firstIndex* through *lastIndex*.

Both functions reset out-of-range indices to be in-range—that is, to the index of the first or last line.

See also: BFont::GetHeight()

LineWidth()

float **LineWidth(**int32 *index* = 0**)** const

Returns the width of the line at *index*—or, if no index is given, the width of the first line. The value returned is the sum of the widths (in coordinate units) of all the characters in the line, from the first through the last, including tabs and spaces. Line indices begin at 0.

If the *index* passed is out-of-range, it's reinterpreted to be the nearest in-range index—that is, as the index to the first or the last line.

See also: BFont::StringWidth()

MakeEditable() , IsEditable()

void **MakeEditable(**bool *flag* = true**)**
bool **IsEditable(**void**)** const

The first of these functions sets whether the user can edit the text displayed by the BTextView; the second returns whether or not the text is currently editable. Text is editable by default.

When text is editable but not selectable, the user can enter and delete text at the insertion point, but can't select text to make changes to more than one character at a time.

See also: `MakeSelectable()`

MakeFocus()

virtual void **MakeFocus(**bool *flag* = true**)**

Overrides the BView version of `MakeFocus()` to highlight the current selection when the BTextView becomes the focus view (when *flag* is `true`) and to unhighlight it when the BTextView no longer is the focus view (when *flag* is `false`). However, the current selection is highlighted only if the BTextView's window is the current active window.

This function is called for you whenever the user's actions make the BTextView become the focus view, or force it to give up that status.

See also: `BView::MakeFocus()`, `MouseDown()`

MakeResizable() , IsResizable()

void **MakeResizable(**bool *resizable*, BView **containerView* = NULL**)**

bool **IsResizable(**void**)** const

`MakeResizable()` gives the BTextView the ability to automatically resize itself to fit its contents if the *resizable* flag is `true`, and takes away that ability if the flag is `false`. `IsResizable()` returns whether the BTextView is currently resizable.

The frame rectangle and text rectangle of a resizable BTextView automatically grow and shrink to exactly enclose all the characters entered by the user. The object should display just a single line of text (the resizing is horizontal); if the *resizable* flag is `true`, `MakeResizable()` turns off line wrapping. The text can be aligned to the left, right, or center of the text rectangle.

The *containerView* is a view that draws a border around the text (like a BScrollView object) and is the parent of the BTextView; it's the view that's resized to fit the text. The BTextView's resizing mode should be such that it will be resized in tandem with the container (for example, `B_FOLLOW_LEFT_RIGHT` or `B_FOLLOW_ALL_SIDES`). However, if the *containerView* is `NULL`, as it is by default, the BTextView itself is resized to fit the text.

If the *resizable* flag is `false`, the *containerView* argument is ignored.

This resizing mechanism is an alternative to the automatic resizing behavior provided in the BView class. It triggers resizing on the user's entry of text, not on a change in the parent

view's size. The two schemes are incompatible; the container view (or the BTextView, if there is no container) should not automatically resize itself when its parent is resized.

See also: `SetAlignment()`

MakeSelectable() , IsSelectable()

void **MakeSelectable(**bool *flag* = true**)**
bool **IsSelectable(**void**)** const

The first of these functions sets whether it's possible for the user to select text displayed by the BTextView; the second returns whether or not the text is currently selectable. Text is selectable by default.

When text is selectable but not editable, the user can select one or more characters to copy to the clipboard, but can't position the insertion point (an empty selection), enter characters from the keyboard, or paste new text into the view.

See also: `MakeEditable()`

MaxBytes() see SetMaxBytes()

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Augments the BView version of **`MessageReceived()`** to handle scripting requests, dropped data, and four editing messages—`B_CUT`, `B_COPY`, `B_PASTE`, and `B_SELECT_ALL`.

If the *message* was dragged and dropped on the BTextView and it contains `B_MIME_TYPE` data under the name "text/plain", this function inserts the new text at the point where it was dropped—but only if `AcceptsDrop()` returns `true` for the *message*.

This function handles `B_CUT`, `B_COPY`, `B_PASTE`, and `B_SELECT_ALL` messages by calling the `Cut()`, `Copy()`, `Paste()`, and `SelectAll()` functions. A BTextView will get these messages, even if the application doesn't send them, when it's the focus view and the user uses the Command-*x*, Command-*c*, Command-*v*, and Command-*a* shortcuts. See "Shortcuts and Menu Items" in the class overview for information on how to set up compatible Cut, Copy, Paste, and Select All menu items.

To inherit this functionality, `MessageReceived()` functions implemented by derived classes should be sure to call the BTextView version.

See also: **`BView::MessageReceived()`**, `AcceptsPaste()`, `BInvoker::SetMessage()`, `BInvoker::SetTarget()`, `InsertText()`

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Selects text, drags text, and positions the insertion point in response to the user's mouse actions. If the BTextView isn't already the focus view for its window, this function calls `MakeFocus()` to make it the focus view.

`MouseDown()` is called for each mouse-down event that occurs inside the BTextView's frame rectangle.

See also: `BView::MouseDown()`, `MakeFocus()`

MouseMoved()

virtual void **MouseMoved(**BPoint *point*, uint32 *transit*, BMessage **message***)**

Responds to `B_MOUSE_MOVED` messages by changing the cursor to the standard I-beam image for editing text whenever the cursor enters the view and by resetting it to the standard hand image when the cursor exits the view. The cursor is changed to an I-beam for text that is selectable or editable, but only if the BTextView is the current focus view in the active window. However, when the cursor moves over the current selection, this function changes it from the I-beam back to the standard hand image. This is done to indicate that it's possible to drag and drop the current selection.

If a *message* is being dragged to the BTextView, this function tests it see whether it contains textual data and tracks it to its destination.

See also: `BView::MouseMoved()`, `AcceptsDrop()`

OffsetAt() see LineAt()

PointAt() see LineAt()

Paste() see Cut()

Pulse()

virtual void **Pulse(**void**)**

Turns the caret marking the current insertion point on and off when the BTextView is the focus view in the active window. `Pulse()` is called by the system at regular intervals.

This function is first declared in the BView class.

See also: `BView::Pulse()`

ResolveSpecifier() see BHandler::ResolveSpecifier()

ScrollToOffset() , ScrollToSelection()

virtual void **ScrollToOffset(**int32 *offset***)**
void **ScrollToSelection(**void**)**

These functions scroll the text so that the character at *offset*—or the character that begins the current selection—is within the visible region of the view. If the BTextView is equipped with scroll bars, the BScrollBar objects are informed so they can update themselves.

See also: **`BView::ScrollTo()`**

Select()

virtual void **Select(**int32 *start*, int32 *finish***)**

Selects the characters from *start* up to *finish*, where *start* and *finish* are offsets into the BTextView's text. If *start* and *finish* are the same, the selection will be empty (an insertion point). See "Offsets" in the class overview for a discussion of the constraints on the offset arguments.

Normally, the selection is changed by the user. This function provides a way to change it programmatically.

If the BTextView is the current focus view in the active window, **`Select()`** highlights the new selection (or displays a blinking caret at the insertion point). However, it doesn't automatically scroll the contents of the BTextView to make the new selection visible. Call **`ScrollToSelection()`** to be sure that the user can see the start of the selection.

See also: **`Text()`**, **`GetSelection()`**, **`ScrollToSelection()`**, **`GoToLine()`**, **`MouseDown()`**

SelectAll()

void **SelectAll(**void**)**

Selects the entire text of the BTextView, and highlights it if the BTextView is the current focus view in the active window.

See also: **`Select()`**

SetAlignment() , Alignment()

void **SetAlignment(**alignment *where***)**

alignment **Alignment**(void) const

These functions set the way text is aligned within the text rectangle and return the current alignment. Three settings are possible:

B_ALIGN_LEFT	Each line is aligned at the left boundary of the text rectangle.
B_ALIGN_RIGH T	Each line is aligned at the right boundary of the text rectangle.
B_ALIGN_CEN TER	Each line is centered between the left and right boundaries of the text rectangle.

The default is **B_ALIGN_LEFT**.

SetAutoindent() , DoesAutoindent()

void **SetAutoindent**(bool *flag*)
bool **DoesAutoindent**(void) const

These functions set and return whether a new line of text is automatically indented the same as the preceding line. When set to `true` and the user types Return at the end of a line that begins with tabs or spaces, the new line will automatically indent past those tabs and spaces to the position of the first visible character.

The default value is `false`.

SetColorSpace() , ColorSpace()

void **SetColorSpace**(color_space *space*)
color_space **ColorSpace**(void) const

These functions set and return the color space of the offscreen bitmap that buffers the drawing the BTextView does. The default color space is **B_CMAP8**.

See also: the BBitmap class

SetFontAndColor() , GetFontAndColor()

void **SetFontAndColor**(int32 *start*, int32 *finish*, const BFont **font*, uint32 *properties* = B_FONT_ALL, rgb_color **color* = NULL) void **SetFontAndColor**(const BFont **font*, uint32 *properties* = B_FONT_ALL, rgb_color **color* = NU
void **GetFontAndColor**(int32 *offset*, BFont **font*, rgb_color **color* = NULL) const void **GetFontAndColor**(BFont **font*, uint32 **sameProperties*, rgb_color **color* = NULL, bool **sameColor* = NULL) const

These functions set and get the font and color used to display the text. If the BTextView supports multiple character formats, `SetFontAndColor()` sets the font and color of the characters bounded by the *start* and *finish* offsets. If no offsets are given, it sets the font and color of the current selection. However, if multiple character formats are not supported, `SetFontAndColor()` ignores the offsets and formats the entire text.

`SetFontAndColor()` works like BView's `SetFont()` function. It sets the font to the attributes of the *font* BFont object that are enumerated by the *properties* mask. The mask is formed by combining the following constants:

`B_FONT_FAMILY_AND_STYLE`
`B_FONT_SIZE`
`B_FONT_SHEAR`
`B_FONT_ROTATION`
`B_FONT_SPACING`
`B_FONT_ENCODING`
`B_FONT_FACE`
`B_FONT_FLAGS`

In addition, `B_FONT_ALL` is a shorthand for all properties of the specified *font*. However, the BTextView modifies the font to ensure that:

- Characters are not rotated.
- Antialiasing is not disabled.
- The spacing mode is `B_BITMAP_SPACING`.
- The character encoding is UTF-8 (`B_UNICODE_UTF8`).

If the *font* argument is `NULL`, the font is not set and the *properties* mask is ignored.

The color of the characters is set by a pointer to an `rgb_color` structure. If the pointer is `NULL`, as it is by default, the color is not set.

`GetFontAndColor()` gets the font and color used to display the character at *offset*. It modifies the *font* BFont object and the *color* `rgb_color` structure so that they describe the font and color of the character.

If an *offset* isn't specified, `GetFontAndColor()` looks at the current selection. It provides a font and color description of the first character of the selection—or the character at the insertion point if the selection is empty. It also modifies that variable that the *sameProperties* argument refers to so that it lists all the font properties that are uniform for all characters in the selection. Similarly, it indicates, in the variable that *sameColor* refers to, whether all the characters in the selection are displayed in the same color.

See also: `BView::SetFont()`

SetMaxBytes() , MaxBytes()

void **SetMaxBytes(**int32 *max***)**
int32 **MaxBytes(**void**)** const

These functions set and return the maximum number of bytes that the BTextView can accept. The default is the maximum number of bytes that can be designated by a signed 32-bit integer, a number sufficiently large to accommodate all uses of a BTextView. Use this function only if you need to restrict the number of characters that the user can enter in a text field.

Note that these functions count bytes, not characters.

SetRunArray() , RunArray()

void **SetRunArray(**int32 *start*, int32 *finish*, const text_run_array **runs***)*
text_run_array ***RunArray(**int32 *start*, int32 *finish*, int32 **length* = NULL**)**

These functions set and return the font and color formats of all the characters bounded by the *start* and *finish* offsets. The formats are described by a **text_run_array** structure, which has the following fields:

int32 **count**
The number of `text_run` structures in the array.

text_run **runs**[1]
A structure describing the font and color formats in effect at a particular offset in the BTextView's text.

The **text_run** structure describes a run of characters that share the same font and color formats. It has three fields:

int32 **offset**
An offset to the first byte of a character in the text buffer. The text run begins with this character; it continues until another run begins.

BFont **font**
The font that's used to display the run of characters beginning at the specified offset.

rgb_color **color**
The color that's used to display the run of characters beginning at the specified offset.

The first offset of the first **text_run** in the array passed to **SetRunArray()** should be 0; the array returned by **RunArray()** also begins at offset 0.

If the BTextView doesn't support multiple character formats, **SetRunArray()** ignores the *start* and *finish* offsets and sets the entire text to the font and color of the first **text_run** in the array. Similarly, **RunArray()** returns a **text_run_array** with one **text_run** describing the entire text.

RunArray() returns a pointer to memory that it allocated (using malloc()). It puts the number of bytes that it allocated in the variable that the *length* argument points to. Although RunArray() allocated the memory, the caller is responsible for freeing it when the returned text_run_array is no longer needed.

SetRunArray() doesn't assume responsibility for the *runs* data it's passed; it's up to the caller to free it.

See also: SetFontAndColor()

SetStylable() , IsStylable()

void **SetStylable(**bool *stylable***)**
bool **IsStylable(**void**)** const

SetStylable() sets whether the BTextView permits multiple character formats. If the *stylable* flag is `true`, the functions that set the font and color of the text can apply to particular characters in the text buffer. If the flag is `false`, those functions apply only to the entire text. When SetStylable() is called to turn off support for multiple formats, all the text is reformatted in the font and color of the first character.

IsStylable() returns whether multiple formats are permitted. By default, they're not.

See also: SetFontAndColor(), SetRunArray()

SetTabWidth() , TabWidth()

void **SetTabWidth(**float *width***)**
float **TabWidth(**void**)** const

These functions set the distance between tab stops to *width* coordinate units and return the current tab width. Tabs cannot be removed nor can they be individually set; all tabs have a uniform width. The default tab width is 28.0 coordinate units.

SetText()

void **SetText(**const char **text*, int32 *length*, const text_run_array **runs* = NULL**)**
void **SetText(**const char **text*, const text_run_array **runs* = NULL**)**
void **SetText(**BFile **file*, int32 *offset*, int32 *length*, const text_run_array **runs* = NULL**)**

Removes any text currently in the BTextView and copies new text from a *text* buffer or from a *file* to replace it. This function copies *length* bytes of text from the buffer—or all the bytes in the buffer, up to the null character, if a *length* isn't specified. Or it copies *length* bytes from the

file beginning at the *offset* byte. If the *text* or *file* is `NULL` or *length* is 0, it empties the BTextView without replacing the text.

If a *runs* `text_run_array` is provided, it will be used to set the font and color formats of the new text—provided that the BTextView permits multiple character formats. If not, the *runs* array is ignored.

The BTextView doesn't assume ownership of the *text* buffer, the *file*, or the *runs* array; you can delete them when `SetText()` returns.

Text taken from a file is inserted directly into the text, bypassing the `InsertText()` function. In other words, you won't receive an `InsertText()` notification for text taken from a file.

This function is typically used to set the text initially displayed in the view. If the BTextView is already attached to a window, it's updated to show its new contents.

See also: `Text()`, `TextLength()`

SetTextRect() , TextRect()

void **SetTextRect(**BRect *rect***)**
BRect **TextRect(**void**)** const

`SetTextRect()` makes *rect* the BTextView's text rectangle—the rectangle that locates where text is placed within the view. This replaces the text rectangle originally set in the BTextView constructor. The layout of the text is recalculated to fit the new rectangle, and the text is redisplayed.

`TextRect()` returns the current text rectangle.

See also: the BTextView constructor

SetWordWrap() , DoesWordWrap()

void **SetWordWrap(**bool *flag***)**
bool **DoesWordWrap(**void**)** const

These functions set and return whether the BTextView wraps lines on word boundaries, thus pushing entire words that don't fit at the end of a line to the next line. When word wrapping is turned on, the BTextView calls `CanEndLine()` to determine exactly where a line can break. If word wrapping is off, lines break only on a newline character (where the user types Return).

By default, word wrapping is turned on (`DoesWordWrap()` returns `true`).

See also: `SetTextRect()`, `CanEndLine()`

TabWidth() see SetTabWidth()

Text() , GetText() , ByteAt()

const char ***Text**(void) const
void **GetText**(int32 *offset*, int32 *length*, char **buffer*) const
uchar **ByteAt**(int32 *offset*) const

These functions reveal the text contained in the BTextView.

Text() returns a pointer to the text, which may be a pointer to an empty string if the BTextView is empty. The returned pointer can be used to read the text, but not to alter it (use **SetText()**, **Insert()**, **Delete()**, and other BTextView functions to do that).

GetText() copies up to *length* bytes of the text into *buffer*, beginning with the byte at *offset*, and adds a null terminator ('0'). Fewer than *length* bytes are copied if there aren't that many between the specified *offset* and the end of the text. This function doesn't make any attempt to ensure that only full character specifications are copied; it's up to the caller to make sure that a character begins at *offset* and that the last byte copied isn't in the middle of a multibyte character. The results won't be reliable if the *offset* is out-of-range.

ByteAt() returns the byte located at *offset*. The offset doesn't have to be to the first byte of a character.

The pointer that **Text()** returns is to the BTextView's internal representation of the text. When it returns, the text string is guaranteed to be null-terminated and without gaps. However, the BTextView may have had to manipulate the text to get it in that condition. Therefore, there may be a performance price to pay if **Text()** is called frequently. If you're going to copy the text, it's more efficient to have **GetText()** do it for you. If you're going to index into the text, it may be more efficient to call **ByteAt()**.

The pointer that **Text()** returns may no longer be valid after the user or the program next changes the text. Even if valid, the string may no longer be null-terminated and gaps may appear.

See also: **TextLength()**

TextHeight() see LineHeight()

TextLength()

int32 **TextLength**(void) const

Returns the number of bytes of text data the BTextView currently contains—the number of bytes in the string that **Text()** returns (not counting the null terminator).

See also: **Text()**, **SetMaxBytes()**

TextRect() see SetTextRect()

SetDoesUndo() , DoesUndo() , UndoState() , Undo() , undo_state

void **SetDoesUndo(**bool *saylt***)**
bool **DoesUndo(**void**)** const
undo_state **UndoState(**bool **isRedo***)* const
virtual void **Undo(**BClipboard **clipboard***)*
enum **undo_state {}**

These functions and enum comprise BTextView's undo world. The operations that BTextView can undo are described in the `undo_state` table, below. The default undo mechanism is one operation deep: Udoing undoes the previous (undo-able) operation; a second (immediate) undo redoes the operation.

You call `SetDoesUndo()` to enable or disable the undo machinery for this object. By default, undo is enabled. The ability to undo is also controlled by the object's editability (**`MakeEditable()`**), but this is a tautology, since in order to have something to undo you have to have been able to edit the object in the first place.

`DoesUndo()` tells you whether the object is "undoable" as set by `SetDoesUndo()`. It doesn't take editability into consideration.

UndoState() tells you what the previous action was (as represented by the constants listed below). This is the action that will be undone if the object is told to undo. The *isRedo* value that's returned by reference is set to **`true`** if the previous action occurred because of an undo.

Constant	Meaning
`B_UNDO_UNAVAILABLE`	Nothing to undo
`B_UNDO_TYPING`	Text was inserted or deleted
`B_UNDO_CUT`	Selection was cut
`B_UNDO_PASTE`	Text was pasted
`B_UNDO_CLEAR`	Selection was cleared
`B_UNDO_DROP`	Message was dropped

The `Undo()` hook function is called when the BTextView receives a **`B_UNDO`** message. (By default, **`B_UNDO`** is bound to the window's Command+z shortcut; Undo menu items need to set up the binding explicitly.) The *clipboard* argument is a pointer to the **`be_clipboard`**; if you're using a custom clipboard in your BTextView subclass, you should pass along your clipboard when you invoke the inherited version of `Undo()`.

Don't try to do too much in a subclass implementation of Undo(). In particular, you probably won't get too far if you're trying to "broaden" the undo tree.

Undo() see SetDoesUndo()

UndoState() see SetDoesUndo()

WindowActivated()

virtual void **WindowActivated(**bool *flag***)**

Highlights the current selection when the BTextView's window becomes the active window (when *flag* is `true`)—provided that the BTextView is the current focus view—and removes the highlighting when the window ceases to be the active window (when *flag* is `false`).

If the current selection is empty (if it's an insertion point), it's highlighted by turning the caret on and off (blinking it).

The Interface Kit calls this function for you whenever the BTextView's window becomes the active window or it loses that status.

See also: **`BView::WindowActivated()`**, **`MakeFocus()`**

Scripting Support

The BTextView class implements an unnamed suite consisting of the following messages:

The Selection Property

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the current selection.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the current selection.

The current text selection **Using two int32 offsets to the beginning and end of the selection in either the "result" or "data" arrays, these messages convey the current selection of the object in a manner identical to GetSelection() and Select().** **The Text Property**	**Message**	**Specifiers**	**Meaning**
B_COUNT_PROPERTY	B_DIRECT_SPECIFIER	Returns the length of the text in bytes.	
B_GET_PROPERTY	B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Returns the text in the specified range in the BTextView.	
B_SET_PROPERTY	B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Removes or inserts text into the specified range in the BTextView.	

The text in the view **The "Text" get and set messages correspond to the methods GetText() and SetText(), using a C string to store the**	**Message**	**Specifiers**	**Meaning**

data. If the B_SET_PROPERTY message lacks a "data" member, the selection is deleted; otherwise, "range" bytes are inserted at offset "index." The values specifying the range are given in byte rather than character offsets. The range counts towards the end of the text, even for B_REVERSE_RANGE_SPECIFIER. **The text_run_array Property**			
B_GET_PROPERTY	B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Returns the style information for the text in the specified range in the BTextView.	
B_SET_PROPERTY	B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Sets the style information for the text in the specified range in the BTextView.	

Font and color information for the text These messages correspond to the methods RunArray() and SetRunArray(), storing the result as a B_RAW_TYPE. As with the "Text" property, the values specifying the range are given in byte rather than character offsets. The range counts towards the end of the text, even for B_REVERSE_RANGE_SPECIFIER.	Field	Type code	Meaning

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

"_text"	`B_STRING_TYPE`	The BTextView's text.
"_align"	`B_INT32_TYPE`	The text `alignment`.
"_tab"	`B_FLOAT_TYPE`	The tab width.
"_col_sp"	`B_INT32_TYPE`	Color space.
"_trect"	`B_RECT_TYPE`	The text rectangle.
"_max"	`B_INT32_TYPE`	The maximum size (*a la* `SetMaxBytes()`).
"_sel" (array)	`B_INT32_TYPE`	index 0: selection start; index 1: selection end
"_dis_ch" (array)	`B_RAW_TYPE`	Disallowed characters.
"_runs"	`B_RAW_TYPE`	Flattened run array.
"_stylable"	`B_BOOL_TYPE`	**`true`** == is stylable.
"_auto_in"	`B_BOOL_TYPE`	**`true`** == autoindent on
"_wrap"	`B_BOOL_TYPE`	**`true`** == word wrapping on
"_nsel"	`B_BOOL_TYPE`	**`true`** == not selectable
"_nedit"	`B_BOOL_TYPE`	**`true`** == not editable

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if word wrapping is off, the "_wrap" field won't be found in the archive.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTextView Function Summary

Function Summary

Function	Description
AcceptsDrop()	Tests whether the drop message is acceptable
AcceptsPaste()	Tests whether the paste message is acceptable
Alignment()	Returns the alignment of the text in the view
AllowChar()	Tests whether BTextView accepts a given character
Archive()	Archives the BTextView into a BMessage
AttachedToWindow()	Provided for implementation
ByteAt()	Returns the byte at a given offset in the text
CanEndLine()	Tests whether a character can end a line of text
Clear()	Removes the selected text
ColorSpace()	Returns the color space of the offscreen rendering bitmap
Copy()	Copies selected text to the clipboard
CountLines()	Counts the lines in the BTextView
CurrentLine()	Returns the current line
Cut()	Moves selected text to the clipboard

Delete()	Removes characters from the BTextView
DeleteText()	Hook called to remove text from the object
DetachedFromWindow()	Provided for implementation
DisallowChar()	Instruct BTextView to reject a given character
DoesAutoindent()	Tests whether autoindent is enabled
DoesUndo()	Tests whether undo is enabled
DoesWordWrap()	Tests whether word wrap is enabled
Draw()	Provided for implementation
FindWord()	Look for a word
FlattenRunArray()	Flattens a text_run_array structure
FrameResized()	Provided for implementation
GetDragParameters()	Hook called when a drag session is initiated
GetFontAndColor()	Returns the font and color used to display the text
GetSelection()	Returns the current selection
GetText()	Returns a run of text in the object
GetTextRegion()	Returns the region containing a given run of characters
GoToLine()	Moves insertion point to a given line
HideTyping()	Makes the typed text invisible or visible

Highlight()	Highlights the given run of characters
Insert()	Inserts characters into the BTextView
InsertText()	Hook called to add text to the object
Instantiate()	Creates a BTextView from a BMessage
IsEditable()	Tests whether the text is editable
IsResizable()	Tests if the object can automatically resize itself
IsSelectable()	Tests whether the user can select text in the view
IsStylable()	Returns whether or not the view accepts multiple character formats
IsTypingHidden()	Tests whether the text is visble
KeyDown()	Provided for implementation
LineAt()	Returns the line number of a given character
LineHeight()	Returns the height of a given line
LineWidth()	Returns the width of a given line
MakeEditable()	Makes the text editable or uneditable
MakeFocus()	Provided for implementation
MakeResizable()	Sets object's ability to automatically resize itself
MakeSelectable()	Sets whether or not the user can select text in the view

MaxBytes()	Returns the maximum number of bytes the view can accept
MouseDown()	Provided for implementation
MouseMoved()	Provided for implementation
OffsetAt()	Returns the text offset of a given line or coordinate location
Paste()	Copies selected text from the clipboard
PointAt()	Returns the coordinate location of a given character
Pulse()	Provided for implementation
RunArray()	Returns the font and color formats for a given run of characters
ScrollToOffset()	Scrolls the text to a given character
ScrollToSelection()	Scrolls the text to the selection
SelectAll()	Selects all the text in the view
Select()	Selects a given run of characters
SetAlignment()	Sets the alignment of the text in the view
SetAutoindent()	Enables or disables autoindent
SetColorSpace()	Sets the color space of the offscreen rendering bitmap
SetDoesUndo()	Enables or disables undo
SetFontAndColor()	Sets the font and color used to display the text
SetMaxBytes()	Sets the maximum number of bytes the view can accept

SetRunArray()	Sets the font and color formats for a given run of characters
SetStylable()	Sets whether or not the view accepts multiple character formats
SetTabWidth()	Sets the distance between tab stops
SetText()	Replaces text in the view
SetTextRect()	Sets the text rectangle
SetWordWrap()	Enables or disables word wrap
TabWidth()	Returns the distance between tab stops
Text()	Returns all the text in the object
TextHeight()	Returns the height of a run of lines
TextLength()	Returns the number of bytes of text in the object
TextRect()	Returns the text rectangle
BTextView()	Constructor
~BTextView()	Destructor
Undo()	Undoes the previous action
UndoState()	Returns information about the previous action
UnflattenRunArray()	Unflattens a text_run_array structure
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSlider

Derived from: public BControl
Declared in: be/interface/Slider.h
Library: libbe.so
Summary: *more...*

A slider is a control that provide a standard way for the user to select a value from a predetermined range; for example, if your application needs to let the user pick a value between 1 and 100, it's easy to create a BSlider object that the user can use to do this. Sliders can be either horizontal or vertical.

BSlider objects also provide a wide assortment of hooks that let you customize the appearance and behavior of your slider controls.

Using BSliders

Creating a basic BSlider is very simple:

```
BSlider(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    int32 minValue, int32 maxValue,
    thumb_style thumbType = B_BLOCK_THUMB,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_FRAME_EVENTS|B_WILL_DRAW | B_NAVIGABLE)
```

- *frame* is the frame rectangle of the BSlider. This is used to determine the position the slider will occupy, but when the BSlider is attached to a window, it will automatically be resized to its preferred size—the bottom and right fields in the BRect are, essentially, ignored.
- *name* is the name of the BSlider.
- *label* is the label that will be displayed above the slider when it's drawn in a window.
- *message* is the message that will be sent when the slider is invoked.
- *minValue* and *maxValue* are the minimum and maximum values the slider can represent.
- *thumbType* is the thumb style, which is either **B_BLOCK_THUMB** or **B_TRIANGLE_THUMB**.
- *resizingMode* and *flags* are the standard BView resizing mode and flags.

```
r.Set(5,5,255,55);
slider = new BSlider(r, "const:slider1", "Construct Speed (%)", NULL,
                0, 140);
```

This creates a slider named "const:slider1" with the label "Construct Speed (%)" above it. The range of possible values for the slider is 0 to 140.

You can add a splash of color to the slider, too. For example, if you want the slider bar to be light blue, you might add:

```
slider->SetBarColor(color);
```

You can also add hash marks to the slider:

```
slider->SetHashMarks(B_HASH_MARKS_BOTTOM);
slider->SetHashMarkCount(10);
```

In this example, SetHashMarks() is used to specify that the hash marks should be below the slider; you can also specify B_HASH_MARKS_TOP to put them above the slider, B_HASH_MARKS_BOTH to put them both above and below the slider, and B_HASH_MARKS_NONE to omit them entirely (which is the default).

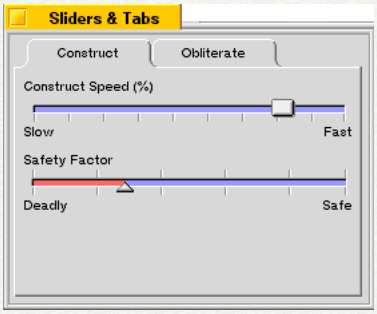
SetHashMarkCount() is called to indicate that you want 10 hash marks spaced evenly across the slider.

Finally, if you want to add labels at the ends of the slider, to indicate the minimum and maximum values, you can use the SetLimitLabels() function:

```
slider->SetLimitLabels("Slow", "Fast");
```

This sets the label for the minimum value (the left end of the slider) to "Slow" and the label for the maximum to "Fast".

The result is the "Construct Speed (%)" slider in the window pictures below. An example of a slider with the B_TRIANGLE_THUMB *thumbType* and hash marks above and below the slider is also shown.



Customizing a BSlider

You can customize the appearance of your BSlider by overriding virtual functions in the BSlider class that are responsible for drawing the slider bar, thumb, focus mark, and hash marks.

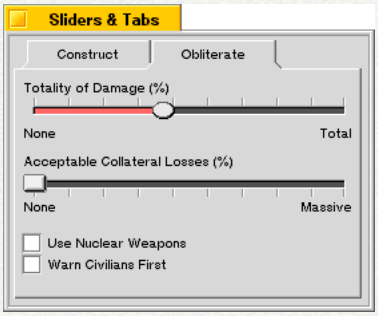
For example, let's say you want your slider's thumb to be round. Just create a new class—let's call it CustomSlider—derived from BSlider, and override the **DrawThumb()** function with code that might look something like this:

```
const rgb_color kWhite = {255,255,255,255};
const rgb_color kWhiteGray = {235,235,235,255);
const rgb_color kDarkGray = {100,100,100,255};
const rgb_color kBlackColor = {0,0,0,255};

void CustomSlider::DrawThumb(void) {
   BRect r;
   BView *v;

   // Get the frame rectangle of the thumb
   // and the offscreen view.

   r = ThumbFrame();
   v = OffscreenView();

   // Draw the black shadow

   v->SetHighColor(kBlackColor);
   r.top++;
   r.left++;
   v->StrokeEllipse(r);

   // Draw the dark grey edge

   v->SetHighColor(kDarkGray);
   r.bottom--;
   r.right--;
   v->StrokeEllipse(r);

   // Fill the inside of the thumb

   v->SetHighColor(kWhiteGrayColor);
   r.InsetBy(1,1);
   v->FillEllipse(r);
}
```

All rendering of a BSlider is done into an offscreen view, which you can get a pointer to by calling the `OffscreenView()` function. This improves performance and makes drawing the slider—especially complicated ones—look much smoother to the user.

This code gets a pointer to the offscreen view, then renders a round, beveled thumb into the offscreen view, filling the thumb's frame rectangle as returned by the `ThumbFrame()` function.

The result is a round thumb that looks like the one in the "Totality of Damage" slider in the picture below:



Hook Functions

- BarFrame()
- DrawBar()
- DrawFocusMark()
- DrawHashMarks()
- DrawSlider()
- DrawLabel()
- DrawThumb()
- GetPreferredSize()
- HashMarksFrame()
- ResizeToPreferred()
- SetBarColor()
- SetBarThickness()
- SetFont()
- SetHashMarks()
- SetHashMarkCount()
- SetKeyIncrementValue()
- SetLabel()
- SetLimitLabels()
- SetModificationMessage()
- SetOrientation()
- SetPosition()
- SetSnoozeAmount()
- SetStyle()
- SetValue()
- ThumbFrame()
- UpdateText()

- **UseFillColor()**
- **ValueForPoint()**

Constructor and Destructor

BSlider()

```
BSlider(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    int32 minValue, int32 maxValue,
    thumb_style thumbType = B_BLOCK_THUMB,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_FRAME_EVENTS | B_WILL_DRAW | B_NAVIGABLE)
BSlider(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    int32 minValue, int32 maxValue,
    orientation posture,
    thumb_style thumbType = B_BLOCK_THUMB,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_FRAME_EVENTS | B_WILL_DRAW | B_NAVIGABLE)
BSlider(BMessage *archive)
```

Initializes the BSlider by passing the appropriate arguments to the BControl constructor. BControl initializes the slider's *label* and assigns it a model *message* that identifies the action that should be carried out when the slider is invoked.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change.

The *minValue* and *maxValue* parameters define the minimum and maximum values to which the slider can be set.

If you want to control whether the slider is horizontal or vertical, use the second form of the constructor, and set the *posture* argument to either **B_HORIZONTAL** or **B_VERTICAL**. If you don't specify this argument, the slider will be horizontal by default.

The *thumbType* argument defines the look of the slider's thumb, and can be either of the following values:

B_BLOCK_THUMB	The thumb is a rectangular block.
B_TRIANGLE_THUMB	The thumb is a triangular pointer.

~BSlider()

```
virtual ~BSlider()
```

Frees memory allocated by the BSlider object.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow**(void)

Augments the BControl version of this function to set the background color of the button so that it matches the background color of its parent. This function also sets the start position of the slider and allocates other memory needed for the slider to function.

See also: `BView::AttachedToWindow()`, `BControl::AttachedToWindow()`

BarColor() see SetBarColor()

BarThickness() see SetBarThickness()

Draw()

virtual void **Draw**(BRect *updateRect*)

Draws the slider by calling `DrawSlider()`.

See also: `BView::Draw()`, `DrawSlider()`

DrawBar() , BarFrame()

virtual void **DrawBar**(void)
virtual BRect **BarFrame**(void) const

`DrawBar()` draws the slider bar. The bar is the narrow region the thumb slides through. The bar should be drawn into the offscreen view.

`BarFrame()` returns the frame rectangle that encloses the slider bar.

These functions can be augmented or replaced by your own versions to alter the appearance of the slider bar.

See also: `OffscreenView()`

DrawFocusMark()

virtual void **DrawFocusMark**(void)

If the slider is in focus, draws a mark indicating that this is the case.

You can override this function to alter the appearance of the slider control.

DrawHashMarks() , HashMarksFrame()

virtual void **DrawHashMarks**(void)
virtual BRect **HashMarksFrame**(void) const

`DrawHashMarks()` draws the hash marks.

`HashMarksFrame()` returns the frame rectangle that encloses the hash marks. If the hash marks are set to **`B_HASH_MARKS_TOP`** or **`B_HASH_MARKS_BOTTOM`**, the rectangle encloses only the area needed to draw the marks. If the setting is **`B_HASH_MARKS_BOTH`**, the hash marks frame rectangle is the bounds rectangle of the entire slider control.

These functions can be augmented or replaced by your own versions to alter the appearance of the hash marks.

DrawSlider()

virtual void **DrawSlider**(void)

Draws the entire slider control by calling the functions responsible for drawing the various parts of the control:

- **`DrawBar()`**

`DrawHashMarks()`

- **`DrawThumb()`**

`DrawFocusMark()`

- **`DrawText()`**

Once the slider has been drawn into the offscreen view, it's copied to its parent window.

See also: **`DrawBar()`**, **`DrawHashMarks()`**, **`DrawThumb()`**, **`DrawFocusMark()`**, **`DrawText()`**

DrawText()

virtual void **DrawText**(void)

Draws the slider's text areas. These are the minimum label, maximum label, and status message.

The minimum and maximum labels can be set using the **SetLimitLabels()** function. The status message is obtained by calling the **UpdateText()** function. If you want there to be a status message, simply override **UpdateText()** to return the string you want drawn as the status message.

In the default implementation of this function, the limit labels are only drawn if both of them have been configured to a value other than **NULL**. If either of them is **NULL**, neither will be drawn.

This function can be augmented or replaced by your own version to alter the appearance or placement of the text.

See also: **SetLimitLabels()**, **UpdateText()**

DrawThumb() , ThumbFrame()

virtual void **DrawThumb**(void)
virtual BRect **ThumbFrame**(void) const

`DrawThumb()` draws the slider's thumb. If you choose to reimplement this function, you should call **Style()** to determine whether to draw a block thumb or a triangle thumb.

`ThumbFrame()` returns the frame rectangle that encloses the thumb.

These functions can be augmented or replaced by your own versions to alter the appearance of the thumb.

FrameResized()

virtual void **FrameResized**(float **width*, float **height*)

Augments the BControl version of **FrameResized()** to adjust the offscreen view and bitmap used for rendering the slider.

GetPreferredSize() , ResizeToPreferred()

virtual void **GetPreferredSize**(float **width*, float **height*)
virtual void **ResizeToPreferred**(void)

`GetPreferredSize()` calculates the width and height of the area needed to render the complete slider control, given the current configuration of the control's various settings.

The *width* returned by this function is the width of the slider control. The *height* is calculated by taking the height of the slider bar itself, adding two times the height of the hash marks, then adding room for the status text (if there is any) and limit strings (if there are any).

Note that the height always includes room for hash marks both above and below the slider, even if the slider has no hash marks or only one of the two sets of marks.

`ResizeToPreferred()` resizes the slider control to be the preferred size.

You can override these functions to alter the preferred size of the slider; this is particularly important if you override other functions to change the appearance of the slider.

- HashMarks() see SetHashMarks()
- HashMarkCount() see SetHashMarkCount()

- KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes***)**

Augments the inherited version of **`KeyDown()`** to let the up and right arrow keys increment the value of the slider, and the down and left arrow keys to decrement the value of the slider.

See also: **`BControl::Invoke()`**, **`BView::KeyDown()`**

- MaxLimitLabel() see SetLimitLabels()
- MinLimitLabel() see SetLimitLabels()
- ModificationMessage() see SetModificationMessage()

- MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **`MouseDown()`** to track mouse the mouse when the button is pressed. A single click causes the thumb to immediately reposition itself to the clicked location, and a click and drag motion causes the slider to follow the mouse cursor until the button is released.

If a modification message has been established, it is sent repeatedly while the mouse button is down. This can be used, for example, to let the changes to the value of the slider be instantly reflected in an onscreen display. When the mouse button is released and the slider has been set to its resting position, the slider's model message is sent.

See also: **`BView::MouseDown()`**, **`BControl::Invoke()`**, **`BControl::SetTarget()`**, **`ModificationMessage()`**, **`SetModificationMessage()`**

- OffscreenView()

BView ***OffscreenView(**BView ***view)**

Returns a pointer to the offscreen BView in which the slider is rendered prior to being copied to the screen. Because the slider is a complex graphical construct, it's rendered offscreen, then copied onto the screen. This function is provided to make it possible to further customize the appearance of a custom slider control.

Orientation() see SetOrientation()

ResizeToPreferred() see GetPreferredSize()

SetBarColor() , BarColor()

virtual void **SetBarColor(**rgb_color *bar_color***)**

virtual rgb_color **BarColor(**void**)** const

`SetBarColor()` sets the color of the slider bar.

`BarColor()` returns the slider bar's color.

SetBarThickness() , BarThickness()

virtual void **SetBarThickness(**float *thickness***)**

float **BarThickness(**void**)** const

`SetBarThickness()` sets the slider bar's thickness and **`BarThickness()`** returns the current thickness.

The slider bar's thickness determines how many pixels across the slider bar is. If no thickness is defined, the bar occupies the entire unused width or height of the slider's bounding rectangle.

SetFont()

virtual void **SetFont(**const BFont ***font**, uint32 *properties* = B_FONT_ALL**)**

Sets the font used for the slider's labels to *font*. The *properties* flags indicate which properties of the *font* should be used, and which should remain unchanged.

SetHashMarks() , HashMarks()

virtual void **SetHashMarks(**hash_mark_location *where***)**

hash_mark_location **HashMarks(**void**)** const

`SetHashMarks()` sets the placement of the hash marks. This value can be one of the following:

B_HASH_MARKS_NONE	The slider should have no hash marks.
B_HASH_MARKS_TOP	The slider should have hash marks above the slider bar. For horizontal sliders only.
B_HASH_MARKS_BOTTOM	The slider should have hash marks below the slider bar. For horizontal sliders only.
B_HASH_MARKS_BOTH	The slider should have hash marks above and below the slider bar.
B_HASH_MARKS_LEFT	The slider should have has marks to the left of the slider bar. For vertical sliders only.
B_HASH_MARKS_RIGHT	The slider should have has marks to the right of the slider bar. For vertical sliders only.

`HashMarks()` returns the current placement of the hash marks.

SetHashMarkCount() , HashMarkCount()

virtual void **SetHashMarkCount(**int32 *hash_mark_count***)**
int32 **HashMarkCount(**void**)** const

`SetHashMarkCount()` sets the number of hash marks that will be displayed above and/or below the slider.

`HashMarkCount()` returns the number of hash marks currently set up.

SetKeyIncrementValue() , KeyIncrementValue()

virtual void **SetKeyIncrementValue(**int32 *increment_value***)**
int32 **KeyIncrementValue(**void**)** const

`SetKeyIncrementValue()` sets the amount by which the slider's value changes when the keyboard is used to move the thumb.

`KeyIncrementValue()` returns the amount by which the slider's value changes when the keyboard is used to move the thumb.

SetLimitLabels() , MinLimitLabel() , MaxLimitLabel()

virtual void **SetLimitLabels(**const char **minLabel*, const char **maxLabel***)**
const char ***MinLimitLabel(**void**)** const
const char ***MaxLimitLabel(**void**)** const

`SetLimitLabels()` sets the labels used to mark the minimum and maximum values on the slider.

`MinLimitLabel()` and `MaxLimitLabel()` return the current strings used to mark the minimum and maximum values on the slider.

SetModificationMessage() , ModificationMessage()

virtual void **SetModificationMessage(**BMessage **message***)**
BMessage ***ModificationMessage(**void**)** const

`SetModificationMessage()` sets the model message that's sent while the mouse is being tracked. The modification message is sent repeatedly as long as the mouse button is held down after being initially clicked inside the slider.

`ModificationMessage()` returns the modification message currently set up.

See also: `MouseDown()`

SetOrientation() , Orientation()

virtual void **SetOrientation(**orientation *posture***)**
orientation **Orientation(**void**)** const

`SetOrientation()` sets the slider's orientation and `Orientation()` returns its current orientation. The possible values are `B_HORIZONTAL` and `B_VERTICAL`.

SetPosition() , Position()

virtual void **SetPosition(**float **position***)**
virtual float **Position(**void**)** const

`SetPosition()` sets the value of the slider given a value between 0.0 and 1.0, where 0.0 is the minimum value of the slider, and 1.0 is the maximum value. This lets you set the slider to a relative position without having to look up the maximum and minimum values of the slider.

`Position()` returns the slider value scaled to the range 0.0 to 1.0.

SetSnoozeAmount() , SnoozeAmount()

virtual void **SetSnoozeAmount(**int32 *snooze_time***)**
int32 **SnoozeAmount(**void**)** const

`SetSnoozeAmount()` sets the time (in microseconds) that the slider snoozes for between updates while the mouse is being tracked.

`SnoozeAmount()` returns the time that the slider snoozes while the mouse is being tracked.

SetStyle() , Style()

virtual void **`SetStyle(`**thumb_style *style*`)`
thumb_style **`Style(`**void`)`** const

`SetStyle()` sets the thumb style for the slider. The *style* can be either of the following two values:

`B_BLOCK_THUMB`	The thumb should be a rectangular block.
`B_TRIANGLE_THUMB`	The thumb should be triangular.

`Style()` returns the current thumb style.

SetValue() , Value() , ValueForPoint()

virtual void **`SetValue(`**int32 **value**`)`
virtual int32 **`Value(`**void`)`** const
virtual int32 **`ValueForPoint(`**BPoint *location*`)`** const

`SetValue()` augments the BControl function to pin the value to the maximum and minimum values for the slider and the position of the thumb before calling the inherited function.

`ValueForPoint()` returns the slider value represented by the given screen coordinates.

SnoozeAmount() see SetSnoozeAmount()

Style() see SetStyle()

UpdateText()

virtual char ***`UpdateText(`**void`)`** const

Returns the status message that should be displayed with the slider. By default, this function returns **`NULL`**; if you want a status message to be displayed, simply override this function to return the appropriate string.

The pointer you return is yours; the **`DrawText()`** routine won't dispose of it unless you augment it to do so.

UseFillColor() , FillColor()

virtual void **UseFillColor(**bool *use_fill*, const rgb_color **bar_color* = NULL**)**
virtual bool **FillColor(**rgb_color **bar_color***)** const

`UseFillColor()` sets the color used to fill the area of the slider bar that's "filled." That is, the part of the bar between the minimum value of the slider and the current thumb position. If the *use_fill* parameter is **`true`**, filling this area of the slider with the fill color is turned on. If *use_fill* is **`false`**, the entire slider bar is always drawn in the bar color.

`FillColor()` stores the current fill color in *bar_color*. The boolean value returned by the function is **`true`** if the fill color is being used, **`false`** if not.

Value() see SetValue()

ValueForPoint() see SetValue()

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_mod_msg"	`B_RAW_TYPE`	The flattened modification message.
"_sdelay"	`B_INT32_TYPE`	The snooze amount.
"_bcolor"	`B_INT32_TYPE`	The bar color.
"_fcolor"	B_INT32_TYPE	The fill color.
"_minlbl"	B_STRING_TYPE	The minimum limit label.
"_maxlbl"	B_STRING_TYPE	The maximum limit label.
"_min"	B_INT32_TYPE	The minimum possible value.
"_max"	B_INT32_TYPE	The maximum possible value.
"_incrementvalue"	B_INT32_TYPE	The keyboard navigation increment value.
"_hashcount"	B_INT32_TYPE	The number of hash marks to draw.
"_hashloc"	B_INT16_TYPE	Where to draw the hash marks.
"_sstyle"	B_INT32_TYPE	The thumb style.

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if there is no minimum limit label set, the "_min_lbl" field doesn't exist.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BControl

Derived from: public BView, public BInvoker
Declared in: be/interface/Control.h
Library: libbe.so
Summary: *more...*

BControl is an abstract class for views that draw control devices on the screen. Objects that inherit from BControl emulate, in software, real-world control devices—like the switches and levers on a machine, the check lists and blank lines on a form to fill out, or the dials and knobs on a home appliance.

Controls translate the messages that report generic mouse and keyboard events into other messages with more specific instructions for the application. A BControl object can be customized by setting the message it posts when invoked and the target object that should handle the message.

Controls also register a current value, stored as an `int32` integer that's typically set to `B_CONTROL_ON` or `B_CONTROL_OFF`. The value is changed only by calling `SetValue()`, a virtual function that derived classes can implement to be notified of the change.

Derived Classes

The Interface Kit currently includes six classes derived from BControl—BButton, BPictureButton, BRadioButton, BCheckBox, BColorControl, and BTextControl. In addition, it has two classes—BListView and BMenuItem—that implement control devices but are not derived from this class. BListView and its subclass, BOutlineListView, share an interface with the BList class (of the Support Kit) and BMenuItem is designed to work with the other classes in the menu system. Like BControl, BListView and BMenuItem inherit from the Application Kit's BInvoker class.

As BListView and BMenuItem demonstrate, it's possible to implement a control device that's not a BControl. However, it's simpler to take advantage of the code that's already provided by the BControl class. That way you can keep a simple programming interface and avoid reimplementing functions that BControl has defined for you. If your application defines its own control devices—dials, sliders, selection lists, and the like—they should be derived from BControl.

Hook Functions

`SetEnabled()`

Enables and disables the control device; can be augmented by derived classes to note when the state of the object has changed.

`SetValue()`

Changes the value of the control device; can be augmented to take collateral action when the change is made.

Constructor and Destructor

BControl()

```
BControl(BRect frame,
    const char *name,
    const char *label,
    BMessage *message,
    uint32 resizingMode,
    uint32 flags)
BControl(BMessage *archive)
```

Initializes the BControl by setting its initial value to 0 (`B_CONTROL_OFF`), assigning it a *label*, and registering a model *message* that captures what the control does—the command it gives when it's invoked and the information that accompanies the command. The *label* and the *message* can each be `NULL`.

The *label* is copied, but the *message* is not. The BMessage object becomes the property of the BControl; it should not be deleted, posted, assigned to another object, or otherwise used in application code. The label and message can be altered after construction with the `SetLabel()` and `SetMessage()` functions.

The BControl class doesn't define a `Draw()` function to draw the label or a `MouseDown()` function to post the message. (It does define `KeyDown()`, but only to enable keyboard navigation between controls.) It's up to derived classes to determine how the *label* is drawn and how the *message* is to be used. Typically, when a BControl object needs to take action (in response to a click, for example), it calls the `Invoke()` function, which copies the model message and delivers the copy to the designated target. By default, the target is the window where the control is located, but `SetTarget()` can designate another handler.

Before delivering the message, `Invoke()` adds two data field to it, under the names "when" and "source". These names should not be used for data items in the model.

The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The BControl begins life enabled, and the system plain font is made the default font for all control devices.

See also: the BView constructor, `BLooper::PostMessage()`, `SetLabel()`, `SetMessage()`, `SetTarget()`, `Invoke()`

~BControl()

```
virtual ~BControl()
```

Frees the model message and all memory allocated by the BControl.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BControl object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* doesn't contain data for a BControl object, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Archives the BControl by recording its label, current value, model message, and whether or not it's disabled in the BMessage *archive*.

See also: **`BArchivable::Archive()`**, **`Instantiate()`** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Overrides BView's version of this function to set the BControl's low color and view color so that it matches the view color of its new parent. It also makes the BWindow to which the BControl has become attached the default target for the **`Invoke()`** function, provided that another target hasn't already been set.

`AttachedToWindow()` is called for you when the BControl becomes a child of a view already associated with the window.

See also: **`BView::AttachedToWindow()`**, **`BView::SetFontName()`**, **`Invoke()`**, **`SetTarget()`**

GetSupportedSuites()

virtual status_t **GetSupportedSuites(**BMessage **message***)**

Adds the name "suite/vnd.Be-control" to the *message*. See "Scripting Support" in the class overview and the "Scripting" section in **The Application Kit** chapter for more information.

See also: **`BHandler::GetSupportedSuites()`**

Invoke()

virtual status_t **Invoke(**BMessage **message* = NULL**)**

Copies the BControl's model BMessage and sends the copy so that it will be dispatched to the designated target (which may be a BLooper's preferred handler). The following two pieces of information are added to the copy before it's delivered:

Data	Type code	Description

name		
"when"	B_INT64_TYPE	When the control was invoked, as measured in the number of milliseconds since 12:00:00 AM January 1, 1970.
"source"	B_POINTER_TYPE	A pointer to the BControl object. This permits the message handler to request more information from the source of the message.

These two names shouldn't be used for data fields in the model.

BControl's version of Invoke() overrides the version that the BInvoker class defines. It's designed to be called by derived classes in their MouseDown() and KeyDown() functions; it's not called for you in BControl code. It's up to each derived class to define what user actions trigger the call to Invoke()—what activity constitutes "invoking" the control.

This function doesn't check to make sure the BControl is currently enabled. Derived classes should make that determination before calling Invoke().

See also: BInvoker::Invoke(), SetEnabled()

IsEnabled() see SetEnabled()

IsFocusChanging()

protected: bool **IsFocusChanging**(void) const

Returns `true` if the BControl is being asked to draw because the focus changed, and `false` if not. If the return value is `true`, either the BControl has just become the focus view or it has just lost that status and the Draw() function has been called to update the on-screen display.

This function can be called from inside Draw() to learn whether it's necessary to draw or erase the visible indication that the BControl is the focus view. IsFocus() will return the new status of the view.

See also: MakeFocus()

KeyDown()

virtual void **KeyDown**(const char **bytes*, int32 *numBytes*)

Augments the BView version of KeyDown() to toggle the BControl's value and call Invoke() when the character encoded in *bytes* is either B_SPACE or B_ENTER. This is done to facilitate keyboard navigation and make all derived control devices operable from the keyboard. Some derived classes—BCheckBox in particular—find this version of the function to be adequate. Others, like BRadioButton, reimplement it.

KeyDown() is called only when the BControl is the focus view in the active window. (However, if the window has a default button, B_ENTER events will be passed to that object and won't be

dispatched to the focus view.)

See also: `BView::KeyDown()`, `MakeFocus()`

Label() see SetLabel()

MakeFocus()

virtual void **MakeFocus(**bool *focused* = true**)**

Augments the BView version of this function to call the BControl's `Draw()` function when the focus changes. This is done to aid keyboard navigation among control devices. If the `Draw()` function of a derived class has a section of code that checks whether the object is in focus and marks the on-screen display to show that it is (and removes any such marking when it isn't), the visual part of keyboard navigation will be taken care of. The derived class doesn't have to reimplement `MakeFocus()`. Most of the derived classes implemented in the Interface Kit depend on this version of the function.

When `Draw()` is called from this function, `IsFocusChanging()` returns `true`.

See also: `BView::MakeFocus()`, `KeyDown()`, `IsFocusChanging()`

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Handles scripting messages for the BControl. See "Scripting Support" on page101 for a description of the messages.

See also: `BHandler::MessageReceived()`

ResolveSpecifier()

virtual BHandler ***ResolveSpecifier(**BMessage **message*, int32 *index*, BMessage **specifier*, int32 *command*, const char **property***)**

Resolves specifiers for the "Label" and "Value" properties. See "Scripting Support" in the class overview and the "Scripting" section of **The Application Kit** chapter for more information.

See also: `BHandler::ResolveSpecifier()`

SetEnabled() , IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BControl if the *enabled* flag is `true`, and disables it if *enabled* is `false`. **IsEnabled()** returns whether or not the object is currently enabled. BControls are enabled by default.

While disabled, a BControl won't let the user navigate to it; the **B_NAVIGABLE** flag is turned off if *enabled* is `false` and turned on again if *enabled* is `true`.

Typically, a disabled BControl also won't post messages or respond visually to mouse and keyboard manipulation. To indicate this nonfunctional state, the control device is displayed on-screen in subdued colors. However, it's left to each derived class to carry out this strategy in a way that's appropriate for the kind of control it implements. The BControl class merely marks an object as being enabled or disabled; none of its functions take the enabled state of the device into account.

Derived classes can augment **SetEnabled()** (override it) to take action when the control device becomes enabled or disabled. To be sure that **SetEnabled()** has been called to actually make a change, its current state should be checked before calling the inherited version of the function. For example:

```
void MyControl::SetEnabled(bool enabled)
{
   if ( enabled == IsEnabled() )
      return;
   BControl::SetEnabled(enabled);
   /* Code that responds to the change in state goes here. */
}
```

Note, however, that you don't have to override **SetEnabled()** just to update the on-screen display when the control becomes enabled or disabled. If the BControl is attached to a window, the kit's version of **SetEnabled()** always calls the **Draw()** function. Therefore, the device on-screen will be updated automatically—as long as **Draw()** has been implemented to take the enabled state into account.

See also: the BControl constructor

SetLabel() , Label()

virtual void **SetLabel(**const char ******string***)**

const char ***Label(**void**)** const

These functions set and return the label on a control device—the text that's displayed, for example, on top of a button or alongside a check box or radio button. The label is a null-terminated string.

SetLabel() frees the old label, replaces it with a copy of *string*, and updates the control on-screen so the new label will be displayed to the user—but only if the *string* that's passed differs from the current label. The label is first set by the constructor and can be modified thereafter by this function.

Label() returns the current label. The string it returns belongs to the BControl and may be altered or freed in due course.

See also: the BControl constructor, **BView::AttachedToWindow()**, **BView::SetFontName()**

SetValue() , Value()

virtual void **SetValue(**int32 *value***)**
int32 **Value(**void**)** const

These functions set and return the value of the BControl object.

SetValue() assigns the object a new value. If the *value* passed is in fact different from the BControl's current value, this function calls the object's **Draw()** function so that the new value will be reflected in what the user sees on-screen; otherwise it does nothing.

Value() returns the current value.

Classes derived from BControl should call **SetValue()** to change the value of the control device in response to user actions. The derived classes defined in the Be software kits change values only by calling this function.

Since **SetValue()** is a virtual function, you can override it to take note whenever a control's value changes. However, if you want your code to act only when the value actually changes, you must check to be sure the new value doesn't match the old before calling the inherited version of the function. For example:

```
void MyControl::SetValue(int32 value)
{
   if ( value != Value() ) {
      BControl::SetValue(value);
      /* MyControl's additions to SetValue() go here*/
   }
}
```

Remember that the BControl version of **SetValue()** does nothing unless the new value differs from the old.

Value() see SetValue()

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Makes sure that the BControl, if it's the focus view, is redrawn when the window is activated or deactivated.

See also: **`BView::WindowActivated()`**

Scripting Support

The BControl class implements the suite called "suite/vnd.Be-control" consisting of the following messages:

The Enabled Property

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns whether or not the BControl is currently enabled.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Enables or disables the BControl.

Whether or not the control is enabled **The "Enabled" property corresponds to the methods IsEnabled() and SetEnabled(), using a boolean to store the data.** **The Label Property**	**Message**	**Specifiers**	**Meaning**
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the BControl's label.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the label of the BControl.	

The text label for the control **The "Label" property corresponds to the methods Label() and SetLabel(), using a string to store the data.** **The Value Property**	**Message**	**Specifiers**	**Meaning**
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the BControl's value.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the value of the BControl.	

The value of the control	**Field**	**Type code**	**Meaning**

The "Value" property corresponds to the methods `Value()` and `SetValue()`, using an int32 to store the data.

Archived Fields **The Archive() function adds the following fields to its BMessage argument:**			
"_msg"	B_MESSAGE_TYPE	The BControl's modal invocation message.	
"_label"	B_STRING_TYPE	The BControl's label.	
"_val"	B_INT32_TYPE	The BControl's value.	
"_disable"	B_BOOL_TYPE	true if the control is disabled.	

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the value is 0, the "_val" field won't be found in the archive.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInvoker

Derived from: *none*
Declared in: be/app/Invoker.h
Library: libbe.so
Summary: *more...*

BInvoker is a convenience class that bundles up everything you need to create a handy message-sending package. The BInvoker contains: **(a)** a BMessage, **(b)** a BMessenger (that identifies a target handler), and **(c)** an optional BHandler that handles replies. You set these ingredients, invoke `Invoke()`, and off goes the message to the target. Replies are sent to the reply handler (`be_app` by default).

BInvoker uses `BMessenger::SendMessage()` to send its messages. The invocation is asynchronous, and there's no time limit on the reply.

BInvoker is mostly used as a mix-in class. A number of classes in the Interface Kit—notably BControl—derive from BInvoker.

Constructor and Destructor

BInvoker()

```
BInvoker(BMessage *message, BMessenger messenger)
BInvoker(BMessage *message, const BHandler *handler, const BLooper *looper = NULL)
BInvoker(void)
```

Initializes the BInvoker by setting its message and its messenger.

- The object's BMessage is taken directly as *message*—the object is *not* copied. The BInvoker takes over ownership of the BMessage that you pass in.
- The object's BMessenger is copied from *messenger*, or initialized with *looper* and *handler*. See the BMessenger class for details on how a BMessenger identifies a target.

If you want a reply handler, you have to call **SetHandlerForReply()** after the constructor returns. You can reset the message and messenger through **SetMessage()** and **SetTarget()**.

~BInvoker()

```
virtual ~BInvoker()
```

Deletes the object's BMessage.

Member Functions

BeginInvokeNotify() , EndInvokeNotify()

void **BeginInvokeNotify(**uint32 *kind* = B_CONTROL_INVOKED**)**
void **EndInvokeNotify()**

If for some reason you need to implement a method that emulates an **InvokeNotify()** call inside an **Invoke()** implementation, you should wrap the invocation code in these functions. They set up and tear down an **InvokeNotify()** context.

Command() see SetMessage()

HandlerForReply() see SetHandlerForReply()

Invoke() , InvokeNotify()

virtual status_t **Invoke(**BMessage ***message** = NULL**)**
status_t **InvokeNotify(**BMessage ***message**, uint32 *kind* = B_CONTROL_INVOKED**)**

Invoke() tells the BInvoker's messenger to send a message. If *message* is non-**NULL**, that message is sent, otherwise the object sends its default message (i.e. the BMessage that was passed in the constructor or in **SetMessage()**). The message is sent asynchronously with no time limit on the reply.

Regarding the use of the default message *vs* the argument, a common practice is to reserve the default message as a template, and pass a fine-tuned copy to **Invoke()**:

```
/* Add the current system time to a copy of the default message. */
BMessage copy(invoker.Message());
copy.AddInt64("when", system_time());
invoker.Invoke(&copy);
```

The **InvokeNotify()** function sends the *message* to the target, using the notification change code specified by *kind*. If message is **NULL**, nothing gets sent to the target, but any watchers of the invoker's handler will receive their expected notifications. By default, the *kind* is **B_CONTROL_INVOKED**—the same *kind* sent by a straight **Invoke()**.

In general, you should call **InvokeNotify()** instead of **Invoke()** in new BeOS applications that run under BeOS 5 and later. You can map old code to new like this:

BeOS 4.5.x	**BeOS 5**
Invoke()	InvokeNotify(Message())

Invoke(Message())	InvokeNotify(Message())
Invoke(ModificationMessage())	InvokeNotify(ModificationMessage(), B_CONTROL_MODIFIED)

Invoke() doesn't call **SendNotices()** by default; you'll have to implement code to do it yourself. Here's how:

```
status_t BControl::Invoke(BMessage *msg) {
   bool notify = false;
   uint32 kind = InvokeKind(&notify);

   BMessage clone(kind);
   status_t err = B_BAD_VALUE;

   if (!msg && !notify) {
      // If no message is supplied, pull it from the BInvoker.
      // However, ONLY do so if this is not an InvokeNotify()
      // context -- otherwise, this is not the default invocation
      // message, so we don't want it to get in the way here.
      // For example, a control may call InvokeNotify() with their
      // "modification" message... if that message isn't set,
      // we still want to send notification to any watchers, but
      // -don't- want to send a message through the invoker.
      msg = Message();
   }
   if (!msg) {
      // If not being watched, there is nothing to do.
      if( !IsWatched() ) return err;
   } else {
      clone = *msg;
   }

   clone.AddInt64("when", system_time());
   clone.AddPointer("source", this);
   clone.AddInt32("be:value",fValue);
   clone.AddMessenger(B_NOTIFICATION_SENDER, BMessenger(this));
   if( msg ) err = BInvoker::Invoke(&clone);

   // Also send invocation to any observers of this handler.
   SendNotices(kind, &clone);

   return err;
}
```

RETURN CODES

- **B_OK**. The message was sent.
- **B_BAD_VALUE**. No default message, and no *message* argument.

- *Other errors forwarded from* **BMessenger::SendMessage()**.

InvokeKind()

uint32 **InvokeKind**(bool **notify* = NULL)

Returns the kind passed to **InvokeNotify()**. This should be called from within your implementation of **Invoke()** if you need to determine what kind was specified when **InvokeNotify()** was called. If you care whether **Invoke()** or **InvokeNotify()** was originally called, you can specify a pointer to a bool, *notify*, which is set to **true** if **InvokeNotify()** was called, or **false** if **Invoke()** was called.

This lets you fetch the **InvokeNotify()** arguments from your **Invoke()** code without breaking compatibility with older applications by adding arguments to **Invoke()**.

InvokeNotify() see Invoke()
IsTargetLocal() see SetTarget()
Message() see SetMessage()
Messenger() see SetTarget()

SetHandlerForReply() , HandlerForReply()

virtual status_t **SetHandlerForReply**(BHandler **replyHandler*)
BHandler ***HandlerForReply**(void) const

SetHandlerForReply() sets the BHandler object that handles replies that are sent back by the target. By default (or if *replyHandler* is **NULL**), replies are sent to the BApplication object.

HandlerForReply() returns the object set through **SetHandlerForReply()**. If the reply handler isn't set, this function returns **NULL**, it *doesn't* return **be_app** (even though **be_app** will be handling the reply).

RETURN CODES

- **SetHandlerForReply()** always returns **B_OK**—it doesn't check for validity.

SetMessage() , Message() , Command()

virtual status_t **SetMessage**(BMessage **message*)
BMessage ***Message**(void) const
uint32 **Command**(void) const

SetMessage() sets the BInvoker's default message to point to *message* (the message is *not* copied). The previous default message (if any) is deleted; a **NULL** *message* deletes the previous message without setting a new one. The BInvoker owns the BMessage that you pass in; you mustn't **delete** it yourself.

Message() returns a pointer to the default message, and **Command()** returns its `what` data member. Lacking a default message, the functions return **NULL**.

RETURN CODES

- **SetMessage()** always returns **B_OK**.

SetTarget() , Target() , IsTargetLocal() , Messenger()

virtual status_t **SetTarget(**BMessenger *messenger***)** virtual status_t **SetTarget(**const BHandler ***handler**, const BLooper ***looper** = NULL**)**
BHandler ***Target(**BLooper ****looper** = NULL**)** const
bool **IsTargetLocal(**void**)** const
BMessenger **Messenger(**void**)** const

These functions set and query the BInvoker's target. This is the BHandler to which the object sends a message when **Invoke()** is called. The target is represented by a BMessenger object; you can set the BMessenger as a copy of *messenger*, or initialize it with *looper* and *handler*. See the BMessenger class for details on how a BMessenger identifies a target.

Target() returns the BHandler that's targeted by the object's messenger. If *looper* is non-**NULL**, the BLooper that owns the BHandler is returned by reference. If the target was set as a looper's preferred handler (i.e. **SetTarget(NULL, looper)**), or if the target hasn't been set yet, **Target()** returns **NULL**. The function returns **NULL** for both objects if the target is remote.

IsTargetLocal() returns **true** if the target lives within the BInvoker's application, and **false** if it belongs to some other app.

Messenger() returns a copy of the BMessenger object the BInvoker uses to send messages. If a target hasn't been set yet, the return will be invalid.

RETURN CODES

- **B_OK**. The target was successfully set.
- **B_BAD_VALUE**. The proposed *handler* doesn't belong to a BLooper.
- **B_MISMATCHED_VALUES**. *handler* doesn't belong to *looper*.

SetTarget() doesn't detect invalid BLoopers and BMessengers.

SetTimeout() , Timeout()

status_t **SetTimeout(**bigtime_t *timeout***)**
bigtime_t **Timeout(**void**)** const

`SetTimeout()` sets the timeout that will be used when sending the invocation message to the invoker's target. By default this is `B_INFINITE_TIMEOUT`.

`Timeout()` returns the current setting for this value.

RETURN CODES

- `B_OK`. No error.

- Target() see SetTarget()
- Timeout() see SetTimeout()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInvoker Function Summary

Function Summary

Function	Description
BeginInvokeNotify()	Protected: begins a custom notification context
Command()	Returns the command constant of the current message
EndInvokeNotify()	Protected: ends a custom invocation context
HandlerForReply()	Returns the handler for replies to the BInvoker's message
Invoke()	Sends a BMessage to the target
InvokeKind()	Protected: returns the kind of invocation that is being processed
InvokeNotify()	Sends a BMessage to the target, with a specified notification change code
BInvoker()	Constructor
~BInvoker()	Destructor
IsTargetLocal()	Returns **`true`** if the BInvoker and its target are in the same app
Message()	Returns the current message
Messenger()	Returns a BMessenger for the target of the BInvoker
SetHandlerForReply()	Sets the handler for replies to the BInvoker's message
SetMessage()	Sets the current message
SetTarget()	Sets the BInvoker's target

SetTimeout()	Sets the timeout to use when sending messages to the target
Target()	Returns the target of the BInvoker
Timeout()	Returns the current timeout setting

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BApplication

Derived from: BLooper > BHandler
Mix-in classes: BArchivable
Declared in: be/app/Application.h
Library: libbe.so
Summary: *more...*

The BApplication class defines an object that represents your application, creates a connection to the App Server, and runs your app's main message loop. An app can only create one BApplication object; the system automatically set the global `be_app` object to point to the BApplication object you create.

A BApplication object's most pervasive task is to handle messages that are sent to your app, a subject that's described in detail below. But message handling aside, you can also use your BApplication object to...

- *Control the cursor*. BApplication defines functions that hide and show the cursor, and set the cursor's image. See `SetCursor()`.
- *Access the window list*. You can iterate through the windows that your application has created with `WindowAt()`.
- *Get information about your application*. Your app's signature, executable location, and launch flags can be retrieved through `GetAppInfo()`. Additional information—icons, version strings, recognized file types—can be retrieved by creating an BAppFileInfo object based on your app's executable file. BAppFileInfo is defined in the Storage Kit.

be_app and Subclassing BApplication

Because of its importance, the BApplication object that you create is automatically assigned to the global `be_app` variable. Anytime you need to refer to your BApplication object—from anywhere in your code—you can use `be_app` instead.

Unless you're creating a very simple application, you should subclass BApplication. But be aware that the `be_app` variable is typed as (BApplication *). You'll have to cast `be_app` when you call a function that's declared by your subclass:

```
((MyApp *)be_app)->MyAppFunction();
```

Constructing the Object and Running the Message Loop

As with all BLoopers, to use a BApplication you construct the object and then tell it to start its message loop by calling the `Run()` function. However, unlike other loopers, BApplication's `Run()` doesn't return until the application is told to quit. And after `Run()` returns, you delete the object—it isn't deleted for you.

Typically, you create your BApplication object in your **main()** function—it's usually the first object you create. The barest outline of a typical `main()` function looks something like this:

```
#include <Application.h>

main()[1]
{
   [2]new BApplication("application/x-vnd.your-app-sig")[3];
   /* Further initialization goes here -- read settings, set globals, etc. */
   be_app->Run()[4];
   /* Clean up -- write settings, etc. */
   delete be_app;
}
```

1 The `main()` function doesn't declare *argc* and *argv* parameters (used for passing along command line arguments). If the user passes command line arguments to your app, they'll show up in the **ArgvReceived()** hook function.

2 Why no pointer assignment? The constructor automatically assigns the object to **be_app**, so you don't have to assign it yourself.

3 The string passed to the constructor sets the application's signature. This is a precautionary measure—it's better to add the signature as a resource than to define it here (a resource signature overrides the constructor signature). Use the **FileTypes** app to set the signature as a resource.

4 As explained in the BLooper class, **Run()** is almost always called from the same thread in which you construct the BApplication object. (More accurately, the constructor locks the object, and **Run()** unlocks it. Since locks are scoped to threads, the easiest thing to do is to construct and **Run()** in the same thread.)

Application Messages

After you tell your BApplication to run, its message loop begins to receive messages. In general, the messages are handled in the expected fashion: They show up in BApplication's **MessageReceived()** function (or that of a designated BHandler; for more on how messages are dispatched to handlers, see <x>).

But BApplication also recognizes a set of application messages that it handles by invoking corresponding hook functions. The hook functions are invoked by **DispatchMessage()** so the application messages never show up in **MessageReceived()**.

Overriding the hook functions that correspond to the application messages is an important part of the implementation of a BApplication subclass.

In the table below, the application messages (identified by their command constants) are listed in roughly the order your BApplication can expect to receive them.

Command	Hook function	Description

Constant		
B_ARGV_RECEIVED	ArgvReceived()	Command line arguments are delivered through this message.
B_REFS_RECEIVED	RefsReceived()	Files (entry_refs) that are dropped on your app's icon, or that are double-clicked to launch your app are delivered through this message.
B_READY_TO_RUN	ReadyToRun()	Invoked from within Run(), the application has finished configuring itself and is ready to go. If you haven't already created and displayed an initial window, you should do so here.
B_APP_ACTIVATED	AppActivated()	The application has just become the active application, or has relinquished that status.
B_PULSE	Pulse()	If requested, pulse messages are sent at regular intervals by the system.
B_ABOUT_REQUESTED	AboutRequested()	The user wants to see the app's About... box.

The protocols for the application messages are described in "General Messages".

For more information on the details of when and why the hook functions are invoked, see the individual function descriptions.

A BApplication can also receive the B_QUIT_REQUESTED looper message. As explained in BLooper, B_QUIT_REQUESTED causes Quit() to be called, contingent on the value returned by the QuitRequested() hook function. However, BApplication's implementation of Quit() is different from BLooper's version. Don't miss it.

Other Topics

- *Locking*. As a BLooper, a BApplication mustt be locked before calling certain protected functions. The BApplication locking mechanism is inherited without modification from BLooper.
- *FileTypes settings*. The BApplication object represents your application at run-time. However, some of the characteristics of your app—whether it can be launched more than once, the file types it can open, its icon—are not run-time decisions.

Hook Functions

AboutRequested()

AppActivated()

ArgvReceived()

Pulse()

ReadyToRun()

RefsReceived()

Constructor and Destructor

BApplication()

```
BApplication(const char *signature)
BApplication(const char *signature, status_t *error)
BApplication(BMessage *archive)
```

The constructor creates a new object, locks it, sets the global variable be_app to point to it, and establishes a connection to the Application Server. From this point on, your app can receive messages, although it won't start processing them until you call **Run()**. You can also begin creating and displaying BWindow objects—even before you call **Run()**.

The *signature* constructors assign the argument as the app's application signature. The argument is ignored if a signature is already specified in a resource or attribute of the application's executable (serious apps should always set the signature as both an attribute and a resource). The signature is a MIME type string that must have the supertype "application". For more information on application signatures and how to set them, see <x>.

If you specify *error*, a pointer to a status_t, any error that occurs while constructing the BApplication will be returned in that variable. Alternately, you can call **InitCheck()** to check the results. If an error is returned by the constructor, you shouldn't call **Run()**.

The *archive* constructor is an implementation detail; see the BArchivable class.

~BApplication()

```
virtual ~BApplication()
```

Closes and deletes the application's BWindows (and the BViews they contain), and severs the application's connection to the Application Server.

Never delete a BApplication object while it's running—wait until **Run()** returns. To stop a BApplication (and so cause **Run()** to return), send it a **B_QUIT_REQUESTED** message:

```
be_app->PostMessage(B_QUIT_REQUESTED);
```

Static Functions

AppResources()

```
static BResources *AppResources(void)
```

Returns a BResources object that's configured from your application's executable file. You may read the data in the BResources object, but you're not allowed to write it; see the BResources class for details. The BResources object belongs to the BApplication class and mustn't be freed.

You needn't have a **be_app** object to invoke this function.

Instantiate() see BArchivable::Instantiate()

Member Functions

AboutRequested()

```
virtual void AboutRequested(void)
```

Hook function that's invoked when the BApplication receives a `B_ABOUT_REQUESTED` message, undoubtedly because the user clicked an *About...* menu item. You should implement the function to put a window on-screen that provides the user with information about the application (version number, license restrictions, authors' names, etc).

AppActivated()

```
virtual void AppActivated(bool active)
```

Hook function that's invoked when the application receives a `B_APP_ACTIVATED` message. The message is sent when the app gains or loses active application status. The *active* flag tells you which way the wind blows: **`true`** means your app is now active; **`false`** means it isn't.

The user can activate an app by clicking on or unhiding one of its windows; you can activate an app programmatically by calling `BWindow::Activate()` or `BRoster::ActivateApp()`. (With regard to the latter: This function is called only if the app has an "activatable" window—i.e. a non-modal, non-floating window).

During launch, this function is called after `ReadyToRun()` (provided the app is displaying an activatable window).

Archive() see BArchivable::Archive()

ArgvReceived()

```
virtual void ArgvReceived(int32 argc, char **argv)
```

Hook function that's invoked when the application receives a `B_ARGV_RECEIVED` message. The message is sent if command line arguments are used in launching the app from the shell, or if *argv*/*argc* values are passed to `BRoster::Launch()`.

This function isn't called if there were no command line arguments, or if `BRoster::Launch()` was called without *argv*/*argc* values.

When the app is launched from the shell, **ArgvReceived()**'s arguments are identical to the traditional `main()` arguments: The number of command line arguments is passed as *argc*; the arguments themselves are passed as an array of strings in *argv*. The first *argv* string identifes the executable file; the other strings are the command line arguments proper. For example, this...

```
$ MyApp file1 file2
```

...produces the *argv* array {"./MyApp", "file1", "file2"}.

BRoster::Launch() forwards its *argv* and *argc* arguments, but adds the executable name to the front of the *argv* array and increments the *argc* value.

Normally, the **B_ARGV_RECEIVED** message (if sent at all) is sent once, just before **B_READY_TO_RUN** is sent. However, if the user tries to re-launch (from the command line and with arguments) an already-running app that's set to **B_EXCLUSIVE_LAUNCH** or **B_SINGLE_LAUNCH**, the re-launch will generate a **B_ARGV_RECEIVED** message that's sent to the already-running image. Thus, for such apps, the **B_ARGV_RECEIVED** message can show up at any time.

- CountWindows() see WindowAt()
- DispatchMessage() see BLooper::DispatchMessage()

- GetAppInfo()

```
status_t GetAppInfo(app_info *theInfo) const
```

Returns information about the application. This is a cover for

```
be_roster->GetRunningAppInfo(be_app->Team(), theInfo);
```

See **BRoster::GetAppInfo()** for more information.

- GetSupportedSuites()

```
virtual status_t GetSupportedSuites(BMessage *message)
```

Adds the scripting suite "suite/vnd.Be-application" to message. See "Scripting Suites and Properties" for the suite's properties. Also see **BHandler::GetSupportedSuites()** for more information on how this function works.

- HideCursor() see SetCursor()
- IsCursorHidden() see SetCursor()

IsLaunching()

bool **IsLaunching**(void) const

Returns `true` if the app is still launching. An app is considered to be in its launching phase until **ReadyToRun()** returns. Invoked from within **ReadyToRun()**, `IsLaunching()` returns **true**.

MessageReceived() see BHandler::MessageReceived()

ObscureCursor() see SetCursor()

Pulse() , SetPulseRate()

virtual void **Pulse**(void)

void **SetPulseRate**(**bigtime_t** *rate*)

`Pulse()` is a hook function that's called when the app receives a **B_PULSE** message. The message is sent once every *rate* microseconds, as set in `SetPulseRate()`. The first **Pulse()** message is sent after **ReadyToRun()** returns. If the pulse rate is 0 (the default), the **B_PULSE** messages aren't sent.

You can implement **Pulse()** to do whatever you want (the default version does nothing), but don't try to use it for precision timing: The pulse granularity is no better than 100,000 microseconds.

Keep in mind that `Pulse()` executes in the app's message loop thread along with all other message handling functions. Your app won't receive any `Pulse()` invocations while it's waiting for some other handler function (including **MessageReceived()**) to finish. In the meantime, **B_PULSE** messages will be stacking up in the message queue; when the loop becomes "unblocked", you'll see a burst of `Pulse()` invocations.

Quit() see Run()

QuitRequested()

virtual bool **QuitRequested**(void)

Hook function that's invoked when the app receives a **B_QUIT_REQUESTED** message. As described in the BLooper class (which declares this function), the request to quit is confirmed if `QuitRequested()` returns **true**, and denied if it returns **false**.

In its implementation, BApplication sends **BWindow::QuitRequested()** to each of its BWindow objects. If they all agree to quit, the windows are all destroyed (through **BWindow::Quit()**) and

this **QuitRequested()** returns **true**. But if any BWindow refuses to quit, that window and all surviving windows are saved, and this `QuitRequested()` returns **false**.

Augment this function as you will, but be sure to call the BApplication version in your implementation.

ReadyToRun()

virtual void **ReadyToRun**(void)

Hook function that's called when the app receives a **B_READY_TO_RUN** message. The message is sent automatically during the **Run()** function, and is sent after the initial **B_REFS_RECEIVED** and **B_ARGV_RECEIVED** messages (if any) have been handled. This is the only application message that every running app is guaranteed to receive.

What you do with `ReadyToRun()` is up to you—if your app hasn't put up a window by the time this function is called, you'll probably want to do it here. The default version of **ReadyToRun()** is empty.

RefsReceived()

virtual void **RefsReceived**(BMessage **message*)

Hook function that's called when the app receives a **B_REFS_RECEIVED** message. The message is sent when the user drops a file (or files) on your app's icon, or double clicks a file that's handled by your app. The message can arrive either at launch time, or while your app is already running—use **IsLaunching()** to tell which.

message contains a single field named "refs" that contains one or more **entry_ref** (**B_REF_TYPE**) items—one for each file that was dropped or double-clicked. Do with them what you will; the default implementation is empty. Typically, you would use the refs to create BEntry or BFile objects.

ResolveSpecifier() see BHandler::ResolveSpecifier()

Run() , Quit()

virtual thread_id **Run**(void)

virtual void **Quit**(void)

These functions, inherited from BLooper, are different enough from their parent versions to warrant description.

`Run()` doesn't spawn a new thread—it runs the message loop in the thread that it's called from, and doesn't return until the message loop stops.

`Quit()` doesn't kill the looper thread—it tells the thread to finish processing the message queue (disallowing new messages) at which point `Run()` will be able to return. After so instructing the thread, `Quit()` returns—it doesn't wait for the message queue to empty.

Also, `Quit()` doesn't delete the BApplication object. It's up to you to delete it after `Run()` returns. (However, `Quit()` *does* delete the object if it's called before the message loop starts—i.e. before `Run()` is called.)

SetCursor() , HideCursor() , ShowCursor() , ObscureCursor() , IsCursorHidden()

void **SetCursor(**const void **cursor***)** void **SetCursor(**const BCursor **cursor*, bool *sync* = true**)**
void **HideCursor(**void**)**
void **ShowCursor(**void**)**
void **ObscureCursor(**void**)**
bool **IsCursorHidden(**void**)** const

`SetCursor()` sets the cursor image that's used when this is the active application. You can pass one of the Be-defined cursor constants (**`B_HAND_CURSOR`** and **`B_I_BEAM_CURSOR`**) or create your own cursor image. The cursor data format is described below.

You can also call **`SetCursor()`** passing a BCursor object; specifying *sync* as **`true`** forces the Application Server to immediately resynchronize, thereby ensuring that the cursor change takes place immediately. The default BCursors are **`B_CURSOR_SYSTEM_DEFAULT`** for the hand cursor and **`B_CURSOR_I_BEAM`** for the I-beam text editing cursor.

`HideCursor()` removes the cursor from the screen.

`ShowCursor()` restores it.

`ObscureCursor()` hides the cursor until the user moves the mouse.

`IsCursorHidden()` returns `true` if the cursor is hidden (but not obscured), and `false` if not.

The cursor data format is described in the "Cursor Data Format" section under BCursor.

ShowCursor() see SetCursor()

WindowAt() , CountWindows()

BWindow ***WindowAt(**int32 *index***)** const
int32 **CountWindows(**void**)** const

`WindowAt()` returns the *index*'th BWindow object in the application's window list. If *index* is out of range, the function returns **`NULL`**.

`CountWindows()` returns the number of windows in the window list.

The windows list includes all windows explicitly created by the app—whether they're normal, floating, or modal, and whether or not they're actually displayed—but excludes private windows created by Be classes.

The order of windows in the list has no signficance.

Locking the BApplication object *doesn't* lock the window list. If you need coordinated access to the list, you'll have to provide your own locking mechanism that protects these functions and all BWindow construction and deletion.

Global Variables

be_app

```
BApplication *be_app;
```

be_app is the global variable that represents your BApplication object. You can refer to be_app anywhere you need a reference to the BApplication object that you created. If you want to call a function that's declared by your BApplication subclass, you have to cast be_app to your subclass:

```
((MyApp *)be_app)->MyAppFunction();
```

be_app_messenger

```
BMessenger *be_app_messenger;
```

be_app_messenger is a global BMessenger that targets your be_app object. It's created in the BApplication constructor.

Archived Fields

Field	Type code	Meaning
"mime_sig"	B_STRING_TYPE	Application signature.

Scripting Suites and Properties

Suite: "suite/vnd.Be-application"

"Name"

Command	Specifier	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Gets the name of the application's main thread.

"Window"

Command	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Returns CountWindows().
Not applicable.	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	The message is forwarded to the specified BWi

---

The Be Book The Application Kit The Application Kit Index

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BApplication Function Summary

Function Summary

Function	Description
AboutRequested()	Hook that's called when the user want an About... box.
AppActivated()	Hook called when the application's active state changes
BApplication()	Constructor with app signature
~BApplication()	Destructor
ArgvReceived()	Hook that passes along command line arguments
CountWindows()	Returns the number of windows in the window list
GetAppInfo()	Returns application-specific information
GetSupportedSuites()	Hook called in response to **`B_GET_SUPPORTED_SUITES`** scripting requests
HideCursor()	Removes the cursor from the screen
IsCursorHidden()	Returns cursor visibility state
IsLaunching()	Returns **`true`** if the application is in the process of launching
ObscureCursor()	Hides the cursor until the mouse is moved
Pulse()	Hook called at regular intervals
Quit()	Quits the main message loop
QuitRequested()	Hook called to predicate a quit.

ReadyToRun()	Hook called when the app is finished launching.
RefsReceived()	Hook called when files are double-clicked or dropped on the app icon
Run()	Runs the main message loop
SetCursor()	Sets the cursor for the application
SetPulseRate()	Determines the interval between **`B_PULSE`** messages
ShowCursor()	Restores the cursor after a **`HideCursor()`**
WindowAt()	Returns a window from the window list

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAppFileInfo

Derived from: BNodeInfo
Declared in: be/storage/AppFileInfo.h
Library: libbe.so
Summary: *more...*

BAppFileInfo lets you get and set the signature, supported types, icons, app flags, and version info that's stored in an executable file's attributes and/or resources. The object also knows how to write certain particles of information into the File Type database and, if the executable is the progenitor of a running application, into the app roster (**be_roster**).

Most apps won't ever need to create or use a BAppFileInfo object. Setting an executable's info is best left to the file's creator, through the use of resource data that's compiled into the executable. Even the act of looking at a file's info should be rare for a normal application.

BAppFileInfo objects should *only* be used to examine and set the characteristics of applications and add-ons. Using the object on a non-executable file could corrupt the file.

Initialization

You initialize a BAppFileInfo object by passing it a BFile object that represents an application or add-on. The BFile needn't be open for reading, but it must be open for writing if you want to set information. The BAppFileInfo object doesn't take over ownership of the BFile that you pass it; in particular, deleting a BAppFileInfo doesn't cause the underlying BFile to be deleted.

To initialize a BAppFileInfo to point to the executable of **be_app**, you do this:

```
/* To get app file info for be_app. */
app_info appInfo;
BFile file;
BAppFileInfo appFileInfo;

be_app->GetAppInfo(&appInfo);
file.SetTo(&appInfo.ref, B_READ_WRITE);
appFileInfo.SetTo(&file);
```

Attributes, Resources, and the File Type Database

When you ask a BAppFileInfo object to get some information, it looks in its BFile's attributes; if the information isn't there, it then looks in the file's resources. When you ask it to *set* some information, the info is written as an attribute and also stored as a resource. You can modify

this behavior through `SetInfoLocation()`: You can tell the object to only access the file's attributes, or to only access the resources.

The signature, icons, and supported types that you set through the functions provided here (`SetSignature()`, `SetIcon()`, `SetIconForType()`, `SetSupportedTypes()`) are also recorded in the File Types database, as described in the various functions.

Functions Inherited From BNodeInfo

You should take care when using the following functions (inherited from BNodeInfo):

- `SetPreferredApp()`. **Never set an application's preferred app**; an application is automatically set to be its own preferred app—it won't work otherwise. An add-on's preferred app is usually itself, but it doesn't have to be. For example, you could set an add-on's preferred app to be the server or application that loads the add-on.
- `SetType()`. **Never set the type of an application or add-on**. The type is automatically set to be **`B_APP_MIME_TYPE`** (a platform-dependent value). If you change the type, your application or add-on will still run (probably), but other parts of the system (double-clicked documents, for example) may have a hard time finding it.

Errors

When you ask a BAppFileInfo to retrieve information by reference, the object *doesn't* clear the reference argument if it fails. Because of this, you should always check the error code that's returned by the **`Get...()`** functions.

The common errors that BAppFileInfo functions return are these:

- `B_OK`. Success.
- `B_NO_INIT`. The BAppFileInfo is uninitialized, or its BFile isn't open for writing.
- `B_ERROR`. The BFile was locked when you initialized the BAppFileInfo.

The info-reading and -writing functions may also return the error codes declared by `BNode::ReadAttr()` and `BNode::WriteAttr()`, and by **`BResources::WriteResource()`**.

Constructor and Destructor

BAppFileInfo()

```
BAppFileInfo(void)
BAppFileInfo(BFile *file)
```

The default constructor creates a new, uninitialized BAppFileInfo object. To initialize you have to follow this construction with a call to **SetTo()**.

The BFile version intializes the BAppFileInfo by passing the argument to **SetTo()**. See **SetTo()** for details.

~BAppFileInfo()

```
~BAppFileInfo()
```

Destroys the object. The BFile that was used to initialize this object isn't touched.

Member Functions

GetAppFlags() , SetAppFlags()

status_t **GetAppFlags(**uint32 **flags***)** const
status_t **SetAppFlags(**uint32 *flags***)**

These functions get and set the executable's "app flags." These are the constants that determine whether an executable can only be launched once, whether it runs in the background, and so on. The app flag constants are defined in **be/app/Roster.h**; the flags must include one of the following...

B_SINGLE_LAUNCH
B_MULTIPLE_LAUNCH
B_EXCLUSIVE_LAUNCH

...plus either of these two:

B_BACKGROUND_APP
B_ARGV_ONLY

See the BApplication class for details on the meanings of these constants.

RETURN CODES

See "Functions Inherited From BNodeInfo" for return values.

GetIcon() , SetIcon() , GetIconForType() , SetIconForType()

status_t **GetIcon(**BBitmap **icon***,** icon_size *which***)** const
status_t **SetIcon(**const BBitmap **icon*, icon_size *which***)**
status_t **GetIconForType(**const char **file_type*, BBitmap **icon***,** icon_size *which***)** const
status_t **SetIconForType(**const char **file_type*, const BBitmap **icon*, icon_size *which***)**

GetIcon() and **SetIcon()** get and set the icons that are represent the executable. **GetIconForType()** and **SetIconForType()** get and set the icons that the executable uses when it writes (or otherwise "takes possession of") a file of the given type, identified by *file_type*. You specify which icon you want (large or small) by passing **B_LARGE_ICON** or **B_MINI_ICON** as the *which* argument.

The icon is passed in or returned through the *icon* argument. The bitmap must be the proper size: 32x32 for the large icon, 16x16 for the small one. In BRect lingo, that's BRect(0, 0, 31, 31) and BRect(0, 0, 15, 15). The bitmap's color space must be **B_CMAP8**. For example:

```
BBitmap *bitmap = new BBitmap(BRect(0,0,31,31), B_CMAP8);
appFileInfo.GetIcon(bitmap, B_LARGE_ICON);
```

You can remove an icon by passing **NULL** as the *icon* argument to **SetIcon()**.

To create a BBitmap you must have a **be_app** object ; the object needn't be running.

RETURN CODES

- B_OK. Success
- **B_BAD_VALUE**. (**Get...**) **NULL** BBitmap, or invalid *file_type*.
- **B_BAD_VALUE**. (**Set...**) The bitmap data isn't the proper size.
- See "Functions Inherited From BNodeInfo" for other return values.

GetIconForType() see GetIcon()

GetPreferredApp() , SetPreferredApp()

See "Functions Inherited From BNodeInfo". An application's preferred app must be itself; an add-on is more flexible. For syntax, see **BNodeInfo::SetPreferredApp()**.

GetSignature() , SetSignature()

status_t **GetSignature(**char **signature**) const
status_t **SetSignature(**const char **signature**)

These functions get and set the executable's MIME type signature. The *signature* buffer you pass to **GetSignature()** should be at least **B_MIME_TYPE_LENGTH** characters long; the **SetSignature()** buffer must be no longer than that.

When you set an executable's signature, the signature is installed in the File Type database if it's not there already. The old signature isn't removed from the database.

RETURN CODES

- B_OK. Success.
- **B_ENTRY_NOT_FOUND**. (**Get...**) The executable doesn't have a signature.
- **B_BAD_VALUE**. (**Set...**) *signature* is too long.

- See "Functions Inherited From BNodeInfo" for other return values.

GetSupportedTypes() , SetSupportedTypes()

status_t **GetSupportedTypes(**BMessage **types***)** const
status_t **SetSupportedTypes(**const BMessage **types***)** status_t **SetSupportedTypes(**const BMessage **types*, bool *sync_all***)**

These functions get and set the MIME file types that this executable can read and/or write. The *types* BMessage that you pass in looks like this:

Field	Type	Meaning
"types" []	B_STRING_TYPE	An array of MIME strings.

GetSupportedTypes() copies the types into the "types" field; SetSupportedTypes() reads them from the field. The message's command constants (its *what* value) is ignored.

Here we print an executable's supported types:

```
BMessage msg;
uint32 i=0;
char *ptr;

if (appFileInfo.GetSupportedTypes(&msg) != B_OK)
   /* Handle the error. */

while (msg.FindString("types", i++, &ptr) == B_OK)
   printf("> Supported Type: %sn", ptr);
```

If SetSupportedTypes() names a type that doesn't already appear in the File Type database, the new type is added to the database and its preferred handler is set to the executable that this BAppFileInfo object represents.

SetSupportedTypes() clobbers an executable's existing set of supported types. If you want to augment an executable's supported types, you should retrieve the existing set, add the new ones, and *then* call SetSupportedTypes().

RETURN CODES

- B_OK. Success.
- B_NO_MEMORY. Insufficient memory to copy the types.
- See "Functions Inherited From BNodeInfo" for other return values.

See also: Supports()

GetType() , SetType()

Implementation detail; see "Functions Inherited From BNodeInfo". An executable's default file type (for both applications and add-ons) is **B_APP_MIME_TYPE**. This value mustn't be changed. For syntax, see **BNodeInfo::SetType()**.

GetVersionInfo() , SetVersionInfo()

status_t **GetVersionInfo(**version_info **info*, version_kind *kind***)** const
status_t **SetVersionInfo(**const version_info **info*, version_kind *kind***)**

The functions get and set the application's "version info." The information is recorded in the **version_info** structure:

```
struct version_info {
   uint32 major;
   uint32 middle;
   uint32 minor;
   uint32 variety;
   uint32 internal;
   char short_info[64];
   char long_info[256];
}
```

The field names (and types) provide suggestions for the type of info they want to store.

There are two kinds of version info, as specified by the *kind* argument:

- **B_APP_VERSION_KIND** provides information about this specific app.
- **B_SYSTEM_VERSION_KIND** provides information about the "suite," or other grouping of apps, that this app belongs to.

Again, the uses of the two kinds is up to the app developer—currently, nothing in the BeOS depends on any information being stored in either **version_info** structure.

RETURN CODES

- B_OK. The **version_info** was found or set.
- **B_ENTRY_NOT_FOUND**. (**Get...**) the app doesn't have the requested version info.
- See "Functions Inherited From BNodeInfo" for other return values.

InitCheck()

status_t **InitCheck(**void**)** const

Returns the status of the most recent initialization.

RETURN CODES

- B_OK. The object was successfully initialized.
- **B_BAD_VALUE**. The object is uninitialized.

IsSupportedType() , Supports()

bool **IsSupportedType(**const char **type***)** const
bool **Supports(**BMimeType **type***)** const

Returns **true** if the app supports the given MIME type and **false** if it doesn't. **IsSupportedType()** always returns **true** if the application supports "application/octet-stream", while **Supports()** returns **true** only if *type* is explicitly supported.

See also: **GetSupportedTypes()**, **SetSupportedTypes()**

IsUsingAttributes() see SetInfoLocation()
IsUsingResources() see SetInfoLocation()
SetAppFlags() see GetAppFlags()
SetIcon() see GetIcon()
SetIconForType() see GetIcon()

SetInfoLocation() , IsUsingAttributes() , IsUsingResources()

void **SetInfoLocation(**info_location *loc***)**
bool **IsUsingAttributes(**void**)** const
bool **IsUsingResources(**void**)** const

SetInfoLocation() sets the location where the BAppFileInfo object stores its information. It can store them as either attributes, resources, or both. *loc* takes the following values:

B_USE_ATTRIBUTES
B_USE_RESOURCES
B_USE_BOTH

IsUsingAttributes() and **IsUsingResources()** return **true** if the BAppFileInfo object is storing information in the designated location and **false** otherwise.

SetPreferredApp() see GetPreferredApp()

- SetSignature() see GetSignature()
- SetSupportedTypes() see GetSupportedTypes()

- SetTo()

status_t **SetTo(**BFile ***file*****)**

Initializes the BAppFileInfo object by pointing it to *file*, which must be a valid (initialized) BFile object, and must not be locked. The BFile is not copied, or re-opened by BAppFileInfo. In particular, the BAppFileInfo uses *file*'s file descriptor, and doesn't destroy the BFile object when it (the BAppFileInfo) is destroyed or reinitialized.

The BAppFileInfo object doesn't check to make sure that the file that you pass in really *is* an executable. Passing in a non-executable (a plain file, a directory, etc.) could corrupt the file.

RETURN CODES

- B_OK. The object was successfully initialized.
- **B_BAD_VALUE**. *file* is invalid (uninitialized).

- SetType() see GetType()
- SetVersionInfo() see GetVersionInfo()
- Supports() see IsSupportedType()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNodeInfo

Derived from: *none*
Declared in: be/storage/NodeInfo.h
Library: libbe.so
Summary: *more...*

BNodeInfo provides file type information about a particular node; specifically:

- The (MIME) file type.
- The node's icons, including the node-specific icon that the Tracker displays.
- The "preferred app"; this is the application that's used to access the node's contents.

Except for the Tracker icon, all this information can also be set through the BNodeInfo class. None of the information is passed on to the File Type database; if you want to record a node's file type information with the database, you have to create a BMimeType object (based on the node's file type) and go from there.

Initialization

You initialize a BNodeInfo object by passing it a BNode object. Although you can pass any flavor of node, you typically only care about files; passing a BFile object (or any subclass of BNode) is, of course, acceptable. The BNodeInfo object maintains its own pointer to the BNode you pass in. You don't have to avoid touching the BNode while a BNodeInfo is looking at it (or changing it); the only thing you shouldn't do is delete the BNode.

BNodeInfo doesn't care if the BNode is locked—there's no particular reason to lock the BNode before passing it in, but the BNodeInfo won't balk if you do. If you pass in a BFile object, BNodeInfo *does not* obey the BFile's read/write flags. For example, you can set the node info for a BFile even if you've opened it in read-only mode.

Node Info equals Attributes

The BNodeInfo class does nothing more than look in a node's attributes for the information it sets or gets. The attribute names for the various information particles are given in the function descriptions, below. If you want, you can bypass BNodeInfo and get the node information directly by passing the attribute names to BNode's `ReadAttr()` and `WriteAttr()` functions.

The one exception to this is `GetTrackerIcon()`: This function starts by looking in the node's attributes, but then it goes out hunting if it has to (if the icon isn't found in the attributes).

BAppFileInfo

BNodeInfo has a single subclass: BAppFileInfo. You use a BAppFileInfo object to get more information about a specific executable image (file).

Errors

Unlike most of the other Storage Kit classes, when you ask a BNodeInfo to retrieve some information by reference, the object *doesn't* clear the reference argument if it fails. Because of this, you should always check the error code that's returned by the `Get...` functions.

Constructor and Destructor

BNodeInfo()

```
BNodeInfo(void)
BNodeInfo(BNode *node)
```

The default constructor creates a new, uninitialized BNodeInfo object. To initialize you have to follow this construction with a call to **SetTo()**.

The BNode version intializes the BNodeInfo by passing the argument to **SetTo()**; see **SetTo()** for details (and error codes).

~BNodeInfo()

```
~BNodeInfo()
```

Destroys the object. The BNode object that was used to initialize the object isn't touched.

Member Functions

GetAppHint() , SetAppHint()

```
status_t GetAppHint(entry_ref *app_ref) const
status_t SetAppHint(const entry_ref app_ref)
```

These functions get and set the "app hint" for the node. The app hint is an **entry_ref** that identifies the executable that should be used when launching the node. Of course, the **entry_ref** may not point to an application, or it might point to an application with the wrong signature (and so on)—that's why this is merely a hint.

`GetAppHint()` function initializes the **entry_ref** to the hint recorded in the "BEOS:PPATH" attribute of the node; the argument must be allocated before it's passed in.

SetAppHint() stores the path corresponding to the **entry_ref** in the "BEOS:PPATH" attribute of the node. Since the path to the application is stored instead of its **entry_ref**, the hint will break if the application is subsequently moved.

RETURN CODES

- B_OK. The ref was successfully retrieved or set.
- **B_ENTRY_NOT_FOUND**. The attribute "BEOS:PPATH" doesn't exist. (Get)
- **B_NOT_ALLOWED**. The node is on a read-only volume. (Set)

See also: **BMimeType::GetAppHint()**

GetIcon() , SetIcon() , GetTrackerIcon()

```
virtual status_t GetIcon(BBitmap *icon, icon_size which = B_LARGE_ICON) const
virtual status_t SetIcon(const BBitmap *icon, icon_size which = B_LARGE_ICON)
status_t GetTrackerIcon(BBitmap *icon, icon_size which = B_LARGE_ICON)
static status_t GetTrackerIcon(entry_ref *ref,
    BBitmap *icon,
    icon_size which = B_LARGE_ICON)
```

GetIcon() and **SetIcon()** get and set the icon data that's stored in the node's attributes. You specify which icon you want (large or small) by passing **B_LARGE_ICON** or **B_MINI_ICON** as the *which* argument. The icon is passed in or returned through the *icon* argument. The icon data is copied out of or into the BBitmap object.

The bitmap (if you're calling `SetIcon()`) or icon (for `GetIcon()` and `GetTrackerIcon()`) must be the proper size: 32x32 for the large icon, 16x16 for the small one.

If you want to erase the node's icon, pass **NULL** as the *icon* argument to SetIcon().

The icon attributes are stored as "BEOS: L:STD_ICON" (large icon) and "BEOS: M:STD_ICON" (small, or "mini" icon).

GetTrackerIcon() finds the icon that the Tracker uses to display the node. The **static** version lets you identify the node as an entry_ref. Both versions follow the same ordered path in trying to find the icon:

1. First, it looks in the node's attributes. If the attribute doesn't exist, it...

2. ...gets the node's preferred app (as a signature), and asks the File Type database if that signature declares an icon for this node's file type. If the node doesn't have a preferred app, or if the app doesn't designate an icon for the node's type, the function...

3. ...asks the File Type database for the icon based on the node's file type. If still empty-handed, the function...

4. ...asks the File Type database for the preferred app based on the node's file type, and then asks that app for the icon it uses to display this node's file type. If still nothing, we...

5. ...quit.

The function doesn't tell you which branch of the path it found the icon in.

RETURN CODES

- B_OK. The icon was found or set.
- B_NO_INIT. The BNodeInfo is uninitialized.
- B_BAD_VALUE . The bitmap or icon wasn't the proper size.
- *Attribute errors*. See the error codes for BNode::ReadAttr() and BNode::WriteAttr().

GetPreferredApp() , SetPreferredApp()

status_t **GetPreferredApp(**char **signature*, app_verb *verb* = B_OPEN**)** const
status_t **SetPreferredApp(**const char **signature*, app_verb *verb* **= B_OPEN)**

These functions get and set the node's "preferred app." This is the application that's used to access the node when, for example, the user double-clicks the node in a Tracker window.

- The preferred app is identified by *signature*, a MIME string.
- The **app_verb** argument specifies the type of access; currently, the only **app_verb** is **B_OPEN**.

If a node doesn't have a preferred app, the Tracker looks in the File Type database for an app that can open the node's file type.

The attribute that stores the preferred app is named "BEOS:PREF_APP".

RETURN CODES

- `B_OK`. The preferred app was found or set.
- `B_NO_INIT`. The BNodeInfo is uninitialized.
- `B_BAD_VALUE` (**`Set...`** only). The signature argument is too long (greater than **`B_MIME_TYPE_LENGTH`**).
- *Attribute errors*. See the error codes for `BNode::ReadAttr()` and `BNode::WriteAttr()`.

GetTrackerIcon() see GetIcon()

GetType() , SetType()

virtual status_t **`GetType(`**char *_type_**`)`** const
virtual status_t **`SetType(`**const char *_type_**`)`**

These functions get and set the node's file type. The file type, passed in or returned through *type*, is a MIME string.

The attribute that stores the file type is named "BEOS:TYPE".

RETURN CODES

- `B_OK`. The type was found or set.
- `B_NO_INIT`. The BNodeInfo is uninitialized.
- *Attribute errors*. See the error codes for `BNode::ReadAttr()` and `BNode::WriteAttr()`.

InitCheck()

status_t **`InitCheck(`**void**`)`** const

Returns the status of the most recent initialization.

RETURN CODES

- `B_OK`. The object was successfully initialized.
- `B_NO_INIT`. The object is uninitialized.
- See `SetTo()` for more error codes.

- SetAppHint() see GetAppHint()
- SetIcon() see GetIcon()
- SetPreferredApp() see GetPreferredApp()

- SetTo()

status_t **SetTo(**BNode ***node)**

Initializes the BNodeInfo object by pointing it to *node*, which must be a valid (initialized) BNode object. The BNodeInfo maintains its own BNode pointer: You shouldn't delete *node* while the BNodeInfo is accessing it; other changes to the BNode are permitted, but you may want to avoid such antics. Re-initializing a BNodeInfo doesn't affect the previous BNode object.

RETURN CODES

- **B_OK**. The object was successfully initialized.
- **B_BAD_VALUE**. *node* is uninitialized.

- SetType() see GetType()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNodeInfo Function Summary

Function Summary

Function	Description
GetAppHint()	Returns the application hint for the node
GetIcon()	Returns the node's icons
GetPreferredApp()	Sets the node's preferred application
GetTrackerIcon()	Returns the icons the Tracker uses to display the node
GetType()	Returns the node's file type
InitCheck()	Returns the status of the last initialization
BNodeInfo()	Constructor
~BNodeInfo()	Destructor
SetAppHint()	Sets the application hint for the node
SetIcon()	Sets the node's icons
SetPreferredApp()	Returns the node's preferred application
SetTo()	Initializes the object to another node
SetType()	Sets the node's file type

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMimeType

Derived from: none
Declared in: be/storage/Mime.h
Library: libbe.so
Summary: *more...*

The BMimeType class provides three services:

- **It can parse a MIME string**. It can tell you whether the string is valid, what it's *supertype* component is, and whether it has a *subtype* component. (The MIME string format is described in "Valid MIME Strings."
- **It gives you access to the File Type database**. Given a MIME type, it can look in the database and retrieve that type's icon(s), "preferred handler" application, the filename extensions that correspond to it, and so on.
- **It can regard a MIME string as an application signature**, and so get and set the executable file, the file types, and the document icons that correspond to that signature.

All three services operate on MIME *strings*. In other words, they answer questions such as "Does this *string* have a supertype?", "Is this *string* installed in the database?", and so on. You can get the MIME strings from anywhere: from a file's file type attribute, from and application's signature, from the header of an e-mail message, you can even make them up.

Valid MIME Strings

A valid MIME string takes the form...

- *supertype*/[*subtype*]

...where *supertype* is one of the seven "media" strings:

- text
- application
- image
- audio
- video
- multipart
- message

...and (the optional) *subtype* can be just about anything...Except it can't include spaces or any of these forbidden characters:

- / < > @ , ; : " ( ) [ ] ? =

When you initialize a BMimeType object (through the constructor or **SetTo()** function), you have to tell it what MIME string you want it to represent:

- The string can be supertype-only, or it can be supertype/subtype.
- Currently, the supertype is *not* restricted to the seven types listed above, but you're probably making a mistake if you make up a new, unrecognized supertype.
- Neither the supertype nor the subtype can include any of the forbidden characters.
- The entire string must be no longer than **B_MIME_TYPE_LENGTH** characters long. (That's about 240 characters. More than enough.)

You can check the validity of a MIME string without constructing a BMimeType object by calling the static **IsValid()** function:

```
BMimeType::IsValid("text/qwerty");
```

Constructor and Destructor

BMimeType()

```
BMimeType(void)
BMimeType(const char *MIME_string)
```

Constructs a new BMimeType object and initializes its MIME type to a copy of *MIME_string* (if the argument is given). The rules of validity apply (see "Valid MIME Strings", above). To see if the initialization was successful, call **InitCheck()** after you construct a new BMimeType object.

You can also set the MIME type through the **SetTo()** function.

~BMimeType()

```
virtual ~BMimeType()
```

Frees the object's MIME string and destroys the object.

Member Functions

Contains()

bool **Contains(**const BMimeType **other***)** const

Compares the MIME string with *other*, returning `true` if they are identical. If the object is a supertype and it's the supertype of *other*, then the method returns `true`. Otherwise, it returns `false`.

Delete() see Install()

GetAppHint() , SetAppHint()

status_t **GetAppHint(**entry_ref **app_ref***)** const

status_t **SetAppHint(**const entry_ref **app_ref***)**

These functions get and set the "app hint" for the object's application signature. The app hint is a path that identifies the executable that should be used when launching an application that has this signature. For example, when the Tracker needs to launch an app of type "application/YourAppHere", it asks the database for the application hint. This hint is converted to an **entry_ref** before it is passed to the caller. Of course, the path may not point to an application, or it might point to an application with the wrong signature (and so on)—that's why this is merely a hint.

`GetAppHint()` function initializes the **entry_ref** to the hint recorded in the database; the argument must be allocated before it's passed in.

SetAppHint() copies the path corresponding to the **entry_ref** into the database. *app_ref* should point to an executable file that has the same signature as this object's MIME type.

RETURN CODES

- B_OK. The ref was successfully retrieved or set.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_BAD_VALUE**. (Set) The ref is uninitialized.

See also: **BNodeInfo::GetAppHint()**

GetAttrInfo() , SetAttrInfo()

status_t **GetAttrInfo(**BMessage **info***)** const

status_t **SetAttrInfo(**BMessage **info***)**

These functions use a BMessage to get and set the list of attributes that are typically associated with files of the MIME type. The BMessage must have the following fields:

Field Name	Type	element[0..n]
"attr:name"	B_STRING_TYPE	Each element is the name of one attribute.
"attr:public_name"	B_STRING_TYPE	Each element is the human-readable name of one attribute.
"attr:type"	B_INT32_TYPE	Each element is the type code for the corresponding attribute.
"attr:public"	B_BOOL_TYPE	**true** if the attribute is public, **false** if it's private.
"attr:editable"	B_BOOL_TYPE	**true** if the attribute should be user-editable, **false** if not.

You can actually have any fields you want; it's up to applications to determine which attributes they recognize and which they don't.

Each element in each field describes the next attribute. If a file has three attributes, there should be three elements in each field, one per attribute.

RETURN CODES

- B_OK. No error.
- B_BAD_VALUE. Invalid file.

GetFileExtensions() , SetFileExtensions()

status_t **GetFileExtensions(**BMessage **msg***)** const
status_t **SetFileExtensions(**const BMessage **msg***)**

The database associates a list of file extensions (".*xxx*..." filename appendages) with each file type. If a file is otherwise untyped, clients of the database can figure out its type by matching the file's extension to the lists in the database.

These functions get and set the file extensions that are associated with the object's MIME type.

- If you're getting the extensions, you'll find them copied into your BMessage's "extensions" field (the BMessage must be allocated). They're given as an indexed array of strings (B_STRING_TYPE).
- Similarly, you pass in the extensions by adding strings to the message's "extensions" field.
- The BMessage's `what` field is unimportant.

For example, to retrieve all the extensions that correspond to this object's MIME type, you would do the following:

```
BMessage msg();
uint32 i=0;
char *ptr;

if (mime.GetFileExtensions(&msg) != B_OK)
   /* Handle the error. */

while (true) {
   if (msg.FindString("extensions", i++, &ptr) != B_OK)
      break;
   printf("> Extension: %sn", ptr);
}
```

A given extension can be associated with more than one MIME type.

A **NULL** msg to `SetFileExtensions()` clears the type's extension list.

`SetFileExtensions()` clobbers the existing set of extensions. If you want to augment a type's extensions, you should retrieve the existing set, add the new ones, and *then* call **SetFileExtensions()**.

Also, there's no way to ask the database to give you a set of file types that map to a given extension. To find a type for an extension, you have to get all the installed types with **GetInstalledTypes()** and ask each one for its set of extensions.

RETURN CODES

- **B_OK**. The extensions were found or set.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_NO_MEMORY**. Insufficient memory to copy the extensions.

GetIcon() , SetIcon()

status_t **GetIcon(**BBitmap ***icon*****,** icon_size *which***)** const
status_t **SetIcon(**const BBitmap ***icon***, icon_size *which***)**

GetIcon() and **SetIcon()** get and set the icons that are associated (in the database) with this object's MIME type. You specify which icon you want (large or small) by passing **B_LARGE_ICON** or **B_MINI_ICON** as the *which* argument. The icon is passed in or returned through the *icon* argument. The icon data is copied out of or into the BBitmap object.

The bitmap (if you're calling `SetIcon()`) or icon (if you're calling `GetIcon()`) must be the proper size: 32x32 for the large icon, 16x16 for the small one. Additionally, the bitmap must be in the **B_CMAP8** color space (8-bit color), or the application will crash.

If you want to erase the node's icon, pass **NULL** as the *icon* argument to `SetIcon()`.

RETURN CODES

- **B_OK**. The icon was found or set.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_BAD_VALUE**. The bitmap or icon wasn't the proper size.

GetIconForType() , SetIconForType()

status_t **GetIconForType(**const char *_file_type_, BBitmap *_icon_**,** icon_size _which_**)** const
status_t **SetIconForType(**const char *_file_type_, const BBitmap *_icon_, icon_size _which_**)**

These functions get and set the icons that an application that has this object's MIME type as a signature uses to display the given file type. *file_type* must be a valid MIME string.

The icon is passed in or returned through the *icon* argument:

- If you're getting the icon, the BBitmap must be allocated; the icon data is copied into your BBitmap object.
- If you're setting the icon, the bitmap must be the proper size: 32x32 for the large icon, 16x16 for the small one. In BRect lingo, that's BRect(0, 0, 31, 31) and BRect(0, 0, 15, 15).
- If you're setting the icon, the bitmap must be in the **B_CMAP8** color space (8-bit color), or the application will crash.
- You can remove an icon by passing **NULL** as the *icon* argument to **SetIconForType()**.

RETURN CODES

- **B_OK**. The icon was found or set.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_BAD_VALUE**. (`Get`) **NULL** BBitmap pointer, or *file_type* is invalid.
- **B_BAD_VALUE**. (`Set`) The bitmap data isn't the proper size, or *file_type* is invalid.

GetInstalledTypes() , GetInstalledSupertypes()

static status_t **GetInstalledTypes(**BMessage *_types_**)** static status_t **GetInstalledTypes(**const char *_supertype_, BMessage *_subtypes_**)**
static status_t **GetInstalledSupertypes(**BMessage *_supertypes_**)**

These static functions retrieve all the file types that are currently installed in the database, all the installed subtypes for a given supertype, and all the installed supertypes. The types are

copied into the "types" field of the passed-in BMessage (which must be allocated).

RETURN CODES

- `B_OK`. The types were found.
- `B_BAD_VALUE`. The *supertype* string isn't valid.
- `B_NO_MEMORY`. Insufficient memory to copy the types.

GetLongDescription() , SetLongDescription() , GetShortDescription() , SetShortDescription()

status_t **GetLongDescription(**char **description**) const
status_t **SetLongDescription(**const char **description**)
status_t **GetShortDescription(**char **description**) const
status_t **SetShortDescription(**const char **description**)

Each file type has a couple of human-readable description strings associated with it. Neither description string may be longer than **B_MIME_TYPE_LENGTH** characters.

These functions get and set the long and short description strings. The `Get` functions copy the string into the argument (which must be allocated). The `Set` functions copy the string that the argument points to.

RETURN CODES

- `B_OK`. The description was found or set.
- `B_NO_INIT`. The BMimeType is uninitialized.
- `B_BAD_VALUE`. (`Set`) *description* is too long.
- `B_NO_MEMORY`. Insufficient memory to copy the description.

GetPreferredApp() , SetPreferredApp()

status_t **GetPreferredApp(**char **signature**, app_verb *verb* = B_OPEN**)** const
status_t **SetPreferredApp(**const char **signature**, app_verb *verb* = B_OPEN**)**

These functions get and set the "preferred app" for this object's MIME type. The preferred app is the application that's used to access a file when, for example, the user double-clicks the file in a Tracker window: Unless the file identifies (in its attributes) a "custom" preferred app, the Tracker will ask the File Type database for the preferred app that's associated with the file's type.

- The preferred app is identified by *signature*, a MIME string. A value of `NULL` indicates that there is no preferred app for the MIME type.
- The **app_verb** argument specifies the type of access; currently, the only **app_verb** is **B_OPEN**.

RETURN CODES

- **B_OK**. The preferred app was found or set.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_BAD_VALUE** (**Set...** only). The signature argument is too long (greater than **B_MIME_TYPE_LENGTH**).

GetShortDescription() see GetLongDescription()

GetSupertype() see Type()

GetSupportingApps() , GetWildcardApps()

```
status_t GetSupportingApps(BMessage *msg) const
static status_t GetWildcardApps(BMessage *msg)
```

These functions retrieve a list of applications (identified by signature) that know how to handle the object's MIME type (for `GetSupportingApps()`) or all MIME types (`GetWildCardApps()`). The information is returned in *msg*, which must be allocated by the caller. The *msg* format is:

Field name	Type	Meaning
"applications"	B_STRING_TYPE (array)	The signatures of the application that know how to handle the MIME type. The first *n* applications (where *n* is defined by "be:sub", below) can handle the full type (supertype *and* subtype). The rest of the applications in the array handle the supertype only.
"be:sub"	B_INT32_TYPE	The number of applications in the "applications" array that can handle the object's full MIME type. These applications are listed first in the array. This field is omitted if the object represents a supertype only.
"be:super"	B_INT32_TYPE	The number of applications in the "applications" array that can handle the object's supertype (not counting those that can handle the full type). These applications are listed after the full-MIME type supporters. By definition, the `GetWildcardApps()` function never returns supertype-only apps.

For example, here we print the signatures of the apps that can handle "text/plain" and "text" (without checking for errors):

```
BMessage msg();
```

```
BMimeType mime("text/plain");
int32 subs=0, supers=0, n, hold;
char *ptr;

mime.GetSupportingApps(&msg);
msg.FindInt32("be:subs", &subs);
msg.FindInt32("be:supers", &supers);

for (n = 0; n < subs; n++) {
   msg.FindString("applications", n, &ptr);
   printf("Full support: %sn", ptr);
}
hold = n;
for (n = 0; n < supers; n++) {
   msg.FindString("applications", n+hold, &ptr);
   printf("Supertype support: %sn", ptr);
}
```

If an application supports both the full type *and* the supertype, it will be listed only once in the "applications" array (as a full supporter).

To set the types that an application supports, use **BAppFileInfo::SetSupportedTypes()**. To tell an app to support all types, add "application/octet-stream" to its supported-types list.

See also:

RETURN CODES

- B_OK. The signatures were found.
- **B_BAD_PORT**. No **be_app** found.
- **B_NO_INIT**. The BMimeType is uninitialized.
- **B_NO_MEMORY**. Insufficient memory to copy the signatures.

GetWildcardApps() see GetSupportingApps()

InitCheck()

```
status_t InitCheck(void) const
```

Returns the status of the most recent construction or **SetTo()** call.

RETURN CODES

- *See* **SetTo()**.

Install() , Delete() , IsInstalled()

status_t **Install**(void)
status_t **Delete**(void)
bool **IsInstalled**(void) const

`Install()` adds the object's MIME type to the File Type database. `Delete()` removes the type from the database. `IsInstalled()` tells you if the type is currently installed.

None of these functions affect the object's copy of the MIME type; for instance, deleting a MIME type from the database *doesn't* uninitialize the object.

RETURN CODES

- **B_OK**. The type was successfully added or deleted.
- **B_BAD_VALUE**. The object is uninitialized.

Currently, **Install()** may return a random value if the object is already installed. To avoid confusion, you should call `IsInstalled()` first:

```
if (!mime.IsInstalled())
   mime.Install();
```

IsInstalled() see Install()

IsValid() , IsSupertypeOnly()

static bool I**sValid**(const char **MIME_string*)
bool **IsValid**(void) const
bool **IsSupertypeOnly**(void) const

The static **IsValid()** tests its argument for MIME validity. See "Valid MIME Strings" for the rules. The non-static version checks the validity of the object's MIME string.

IsSupertypeOnly() returns **true** if the object's MIME string doesn't include a subtype.

SetAppHint() see GetAppHint()
SetAttrInfo() see GetAttrInfo()
GetFileExtensions()
GetIcon()
GetIconForType()

GetLongDescription()

- GetPreferredApp()
- GetLongDescription()

SetTo() , Unset()

status_t **SetTo(**const char **MIME_string**)
void **Unset(**void**)**

`SetTo()` initializes this BMimeType object to represent *MIME_string*. The object's previous MIME string is freed; the argument is then copied. The argument can be a full supertype/subtype string, or simply a supertype. In any case, it must pass the validity test described in "Valid MIME Strings".

`Unset()` frees the object's current MIME string, and sets the object's status to **B_NO_INITGetLongDescription()** .

RETURN CODES

These return codes are also returned by the **InitCheck()** function.

- **B_OK**. The initialization was successful.
- **B_NO_INIT**. *MIME_string* is **NULL** or invalid.
- **B_NO_MEMORY**. Not enough memory to allocate a copy of the argument.

See also: **InitCheck()**

StartWatching() , StopWatching()

static status_t **StartWatching(**BMessenger *target***)**
static status_t **StopWatching(**BMessenger *target***)**

`StartWatching()` initiates the MIME monitor, which is used for keeping track of changes to the File Types database. Change notifications will be sent via the BMessenger *target* in a BMessage with the *what* field set to **B_META_MIME_CHANGED**.

Notification messages have the following fields:

Name	Type	Description
be:which	int32	Change bitmap (see below for a list)
be:type	string	MIME type whose database information was changed
be:extra_type	string	Extra MIME field used for some notifications
be:large_icon	bool	For notifications involving icon changes, **true** if the large icon was

	changed; **false** otherwise

"be:which" is a bitmask describing the changes made to the database for MIME type "be:type". The following masks are defined along with the BMimeType methods used to effect the changes they signal:

B_ICON_CHANGED	SetIcon()
B_PREFERRED_APP_CHANGED	SetPreferredApp()
B_ATTR_INFO_CHANGED	SetAttrInfo()
B_FILE_EXTENSIONS_CHANGED	SetFileExtensions()
B_SHORT_DESCRIPTION_CHANGED	SetShortDescription()
B_LONG_DESCRIPTION_CHANGED	SetLongDescription()
B_ICON_FOR_TYPE_CHANGED	SetIconForType()
B_APP_HINT_CHANGED	SetAppHint()

The BMimeType methods are given for illustrative purposes only –– anything that alters the database for a MIME type will also trigger a notification message. The "be:extra_type" field is used only in the **B_ICON_FOR_TYPE_CHANGED** message and indicates the application signature for which the change is valid.

StopWatching() terminates the MIME monitor previously initiated for the given BMessenger.

StartWatching()

Type() , GetSupertype()

const char ***Type**(void) const
status_t **GetSupertype(**BMimeType **super**) const

Type() returns a pointer to the object's MIME string. If the object isn't initialized, this returns a pointer to **NULL**.

GetSupertype()StartWatching() initializes the argument with this object's supertype. (You can then call GetType() on the argument to see the supertype.) *super* must be allocated before it's passed in. If this object isn't initialized, *super* is uninitialized.

RETURN CODES

The errors apply to GetSupertype() only.

- B_OK. Everything's fine.
- B_BAD_VALUE. This object isn't initialized.

SetTo()

Operators

== (comparison)

```
bool operator==(const BMimeType &type) const
bool operator==(const char *type) const
```

Two MIME types are equal if they are both initialized to the same string (without regard to case).

C Functions

create_app_meta_mime

```
status_t create_app_meta_mime(const char *path, int recursive, int synchronous,
    int force)
```

Creates an entry in the File Types database for a specific application, or for all applications:

- To create an entry for a specific application, pass the path to the application in *path*. Information is taken from the app and written into the database.
- To create entries for all applications, pass a **NULL** path.

The *recursive* flag is currently unused.

If *synchronous* is **true**, the function doesn't return until the operation is complete. If it's **false**, the function returns immediately while the operation continues in the background.

If *force* is **true**, entries are created even if they already exist.

RETURN CODES

- B_OKSetTo() . No error.
- B_BAD_VALUE. You can't request recursive operation when a path is specified.

---

get_device_icon

```
status_t get_device_icon(const char *device, void *icon, int32 size)
```

Returns the icons that are associated with the given device. You specify which icon you want (large or small) by passing 32 or 16 as the *size* argument. The icon is then returned through the *icon* argument.

See also: BVolume::GetIcon()

---

update_mime_info

```
int update_mime_info(const char *path, int recursive, int synchronous,
    int force)
```

Updates the MIME information for one or more files, specified by *path*. If *path* is **NULL**, all files are scanned recursively, and the value of *recursive* is ignored.

If *recursive* is **true** and *path* indicates a directory, the directory tree contained by *path* is scanned, with every file in the tree being updated; otherwise, just the file indicated by path is updated.

If *synchronous* is **true**, `update_mime_info()` doesn't return until the update operation is complete. If it's **false**, `update_mime_info()` runs asynchronously, and returns immediately while updating continues in the background.

If *force* is **true**, files are updated even if they've been updated already.

If an error occurs, `update_mime_info()` returns a negative error code; otherwise, it returns `B_OK`.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMimeType Function Summary

Function Summary

Function	Description
==	Tests if two MIME types are identical
Contains()	Tests if the MIME type is a supertype of a given type
Delete()	Removes the type from the File Type database
GetAppHint()	Returns the application hint for an object's application signature
GetAttrInfo()	Returns information about the attributes associated with the MIME type
GetFileExtensions()	Returns the file extensions associated with the MIME type
GetIcon()	Returns the icons associated with the MIME type
GetIconForType()	Returns the icons associated with the MIME type for an application
GetInstalledSupertypes()	Returns all the installed supertypes
GetInstalledTypes()	Returns all the installed file types
GetLongDescription()	Returns the MIME type's long description
GetPreferredApp()	Returns the preferred application for the MIME type

GetShortDescription()	Returns the MIME type's short description
GetSupertype()	Returns the object's supertype
GetSupportingApps()	Returns the MIME type's set of supporting applications
GetWildcardApps()	Returns the applications supporting all MIME types
InitCheck()	Returns the status of the last initialization
Install()	Adds the MIME type to the File Type database
IsInstalled()	Tests whether the type is in the File Type database
IsSupertypeOnly()	Tests whether the MIME type is valid
IsValid()	Tests a type for MIME validity
BMimeType()	Constructor
~BMimeType()	Destructor
SetAppHint()	Sets the application hint for an object's application signature
SetAttrInfo()	Sets the attributes associated with files of this MIME type
SetFileExtensions()	Sets the file extensions associated with the MIME type
SetIcon()	Sets the icons associated with the MIME type
SetIconForType()	Sets the icons associated with the MIME type for an application

SetLongDescription()	Sets the MIME type's long description
SetPreferredApp()	Sets the preferred application for the MIME type
SetShortDescription()	Sets the MIME type's short description
SetTo()	Initializes the object to another MIME string
StartWatching()	Begin monitoring File Types changes
StopWatching()	Stop monitoring File Types changes
Type()	Returns a pointer to the object's MIME string
Unset()	Uninitializes the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Real-Time Allocation Functions

Declared in: be/media/RealtimeAlloc.h **Library:** libmedia.so
Summary: *more...*

Media nodes are highly timing-sensitive creatures. The slightest delay in performing their work can cause drastic problems in media playback or recording quality. Virtual memory, normally of great benefit to users, can work against them when doing media work. A poorly-timed virtual memory hit can cause breaks in media performances.

The realtime memory allocation and locking functions provide a means for nodes to lock down their memory to prevent it from being cached to disk by the virtual memory system. This avoids situations in which the node has to pause while it or its memory is fetched back from the swap file.

The user can use the Media preference application to configure what types of nodes should use locked memory. Nodes should typically use the realtime memory allocation functions instead of `malloc()` and `free()`. `rtm_alloc()` will automatically handle locking the memory if the `B_MEDIA_REALTIME_ALLOCATOR` flag is set, so your node doesn't have to worry about it.

Memory Pools

Real-time memory blocks are allocated in **pools**. Pools are locked into physical RAM if realtime allocators are turned on in the BMediaRoster. While pools are locked in memory, that reduces the amount of physical RAM available to other applications, so don't use real-time blocks unless it's really going to benefit your performance.

When you create a pool, you specify the total size of the pool. This size is the maximum number of bytes that can be allocated from the pool. Memory can then be allocated out of the pool by calling `rtm_alloc()`, and memory blocks in the pool can be freed by calling `rtm_free()`.

The Default Pool

There's a special pool, called the default (or shared) pool. You can allocate memory blocks in this pool if you wish, but it's a scarce resource and as a general rule you should avoid it.

Instead, you should create your own pool and use that; let the media nodes and Media Kit use the default pool for its own purposes.

MALLOC_DEBUG

`rtm_alloc()` responds to the value of `MALLOC_DEBUG`. If the `MALLOC_DEBUG` environment variable is set to anything other than 0, new and freed blocks will be cleared to some junk value. If it's set to an integer greater than 1 it will also do pool, heap, and block validations at opportune times. It's not perfect validation, but it's better than nothing.

Global C Functions

rtm_alloc() , rtm_realloc()

```
void *rtm_alloc(rtm_pool *pool, size_t size)
status_t *rtm_realloc(void **data, size_t newSize)
```

`rtm_alloc()` allocates a block of memory of *size* bytes in the specified *pool*. If you specify **NULL** for *pool*, the memory is allocated in the default pool.

`rtm_realloc()` resizes the block referenced by *data* to be *newSize* bytes long. Since the buffer may have to move to accomodate the new size, the pointer pointed to by data may change. If *data* is **NULL**, this is just like calling `rtm_alloc()` on the default pool.

RETURN CODES

- `B_OK`. No error.
- `B_BAD_VALUE`. The *data* pointer is **NULL**.
- `B_NO_MEMORY`. The pool is full.

rtm_create_pool() , rtm_delete_pool()

```
status_t rtm_create_pool(rtm_pool **outPool, size_t totalSize,
    const char *name = NULL)
status_t rtm_delete_pool(rtm_pool *pool)
```

`rtm_create_pool()` creates a new memory pool. The pool's rtm_pool reference is stored in *outPool*. The pool is capable of holding up to *totalSize* bytes of data, and is given the specified *name*.

If *outPool* is **NULL**, the default pool is created if it doesn't already exist. If it does, **EALREADY** is returned.

`rtm_delete_pool()` deletes the specified *pool*. You can't delete the default pool; passing **NULL** as *pool* will return `B_BAD_VALUE`.

Be sure to delete a pool once you're finished with it.

RETURN CODES

- `B_OK`. No error.
- **EALREADY**. The default pool already exists (`rtm_create_pool()` only).

- B_BAD_VALUE. You can't delete the default pool.
- Area errors.

---

rtm_default_pool()

```
rtm_pool *rtm_default_pool(void)
```

Returns a pointer to the default pool, or **NULL** if it hasn't been initialized yet.

---

rtm_delete_pool() see rtm_create_pool()

---

rtm_free()

```
status_t rtm_free(void *data)
```

Frees the specified block of data.

RETURN CODES

- B_OK. No error.
- B_BAD_VALUE. A **NULL** pointer was given.

---

rtm_phys_size_for() see rtm_size_for()

rtm_realloc() see rtm_alloc()

---

rtm_size_for() , rtm_phys_size_for()

```
status_t rtm_size_for(void *data)
status_t rtm_phys_size_for(void *data)
```

rtm_size_for() returns the size of the specified block of memory.

rtm_phys_size_for() returns the physical size of the specified block of memory. This is the maximum number of bytes the block can grow to without risking rtm_realloc() actually moving the block; the physical size is always greater than or equal to the block size.

RETURN CODES

- B_OK. No error.
- B_BAD_VALUE. The *data* pointer is **NULL**.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Real-Time Allocation Functions

Function Summary

Function Summary

Function	Description
rtm_alloc()	Allocates a block of memory in the specified pool
rtm_create_pool()	Creates a new pool
rtm_default_pool()	Returns a pointer to the default pool
rtm_delete_pool()	Deletes a pool
rtm_free()	Frees a block of memory
rtm_phys_size_for()	Returns the physical size of the specified memory block
rtm_realloc()	Reallocates a block of memory in the specified pool, changing its size
rtm_size_for()	Returns the size of the specified memory block

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The area_malloc Module

Declared in: be/drivers/area_malloc.h
Summary: *more...*

The area_malloc module provides a means for your driver to allocate memory in areas instead of on the heap. It provides `malloc()`, `calloc()`, `realloc()`, and `free()` functions that work just like their POSIX counterparts, except they require a pool argument as their first input.

These functions aren't safe to call from interrupt handlers; they may block on semaphores.

The area_malloc functions are thread-safe in relation to one another, but not in relation to `delete_pool()`. Be sure you don't call `delete_pool()` on the pool you're using until you know none of the other functions might be called. `create_pool()` and `delete_pool()` are safe in relation to each other.

When the last user of the module puts it away, any remaining pools are automatically deleted.

Module Functions

create_pool() , delete_pool()

const void ***create_pool(**uint32 *addressSpec*, size_t *size*, uint32 *lockSpec*, uint32 *protection***)**
status_t **delete_pool(**const void **poolID***)*

`create_pool()` creates a new pool of memory from which to allocate. The parameters are the same as those used by `create_area()`, so you have complete control over the area's characteristics (except for its name). Returns an opaque pool idenfityer, or **`NULL`** if the creation failed. The ability to share resources allocated from the pool is determined by the permissions and protections used to create the area.

`delete_pool()` deletes the pool specified by the opaque *poolID* given. Any pointers returned by the other functions in the module are immediately invalid. Returns `B_OK` if the pool was deleted, otherwise `B_ERROR`.

See also: `create_area()` in the Kernel Kit.

malloc() , calloc() , realloc()

void ***malloc(**const void **poolID*, size_t *size***)**
void ***calloc(**const void **poolID*, size_t *numMembers*, size_t *size***)**
void ***realloc(**const void **poolID*, void **ptr*, size_t *size***)**

`malloc()` allocates a block of *size* bytes and returns a pointer to it.

`calloc()` allocates a block that can contain *numMembers* items of the specified *size* and returns a poiner to it.

`realloc()` resizes the memory block pointed to by *ptr* to the indicated *size*. Resizing a block can require that the memory be relocated, so this function returns the new pointer.

Each of these operations functions in the pool specified by *poolID*.

If there's not enough memory to allocate the requested block, these functions return **`NULL`**.

free()

void **free(**const void **poolID*, void **ptr***)**

Releases the memory block pointed to by *ptr* from the pool specified by *poolID*.

Constants

B_AREA_MALLOC_MODULE_NAME

```
#define B_AREA_MALLOC_MODULE_NAME "generic/area_malloc/v1"
```

The B_AREA_MALLOC_MODULE_NAME constant identifies the area_malloc module; use this constant to open the module.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Drivers Table of Contents

- Device Drivers
- Writing Drivers
- Writing Modules
- Using Modules
- The area_malloc Module
- Driver Settings API
- Constants and Defined Types
- Functions for Drivers & Modules

The Be Book Drivers Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Drivers: Master Index

A

acquire_spinlock()	Functions for Drivers & Modules
add_debugger_command()	Functions for Drivers & Modules
B_ADD_FIXED_DRIVER	Constants and Defined Types
add_timer()	Functions for Drivers & Modules
api_version	Writing Drivers
The area_malloc Module	The area_malloc Module
The area_malloc Module	The area_malloc Module
B_AREA_MALLOC_MODULE_NAME	The area_malloc Module
B_AUDIO_DRIVER_BASE	Constants and Defined Types

B

bytes_per_sector	Constants and Defined Types

C

C Functions	Driver Settings API
call_all_cpus()	Functions for Drivers & Modules
calloc()	The area_malloc Module

cancel_timer()	Functions for Drivers & Modules
close_hook()	Writing Drivers
Code Synchronization	Device Drivers
Constants and Defined Types	Constants and Defined Types
Constants and Defined Types	Constants and Defined Types
Constants	Constants and Defined Types
Constants	The area_malloc Module
control_hook()	Writing Drivers
cpu_status	Functions for Drivers & Modules
create_pool()	The area_malloc Module
Current Driver API Version	Constants and Defined Types
cylinder_count	Constants and Defined Types

D

Defined Types	Constants and Defined Types
Defined Types	Driver Settings API
delete_pool()	The area_malloc Module
deselect_hook()	Writing Drivers
devfs	Device Drivers
device	Constants and Defined Types
Device Drivers	Device Drivers
Device Drivers	Device Drivers
Device Drivers	Device Drivers
Device Hooks	Writing Drivers
device_geometry	Constants and Defined Types

device_hooks	Constants and Defined Types
device_icon	Constants and Defined Types
B_DEVICE_OP_CODES_END	Constants and Defined Types
device_type	Constants and Defined Types
disable_interrupts()	Functions for Drivers & Modules
Disabling Interrupts	Device Drivers
Don't Block	Device Drivers
Don't Preempt	Device Drivers
dprintf()	Functions for Drivers & Modules
Driver Control Opcodes	Constants and Defined Types
Driver Implementation Principles	Device Drivers
Driver Rules	Writing Drivers
Driver Settings API	anon
The Driver Settings API	Device Drivers
Driver Settings API	Driver Settings API
Driver Settings API	Driver Settings API
driver_parameter	Driver Settings API
driver_path	Constants and Defined Types
driver_settings	Driver Settings API

E

B_EJECT_DEVICE	Constants and Defined Types

F

File Systems	Device Drivers
find_device()	Writing Drivers
B_FLUSH_DRIVE_CACHE	Constants and Defined Types
B_FORMAT_DEVICE	Constants and Defined Types
free()	The area_malloc Module
free_hook()	Writing Drivers
Functions Available During Spinlocks	Device Drivers
Functions Available While Interrupts Are Disabled	Device Drivers
Functions for Drivers & Modules	Functions for Drivers & Modules
Functions for Drivers & Modules	Functions for Drivers & Modules

---

G

B_GET_BIOS_DRIVE_ID	Constants and Defined Types
B_GET_BIOS_GEOMETRY	Constants and Defined Types
B_GET_DEVICE_SIZE	Constants and Defined Types
get_driver_boolean_parameter()	Driver Settings API
B_GET_DRIVER_FOR_DEVICE	Constants and Defined Types
get_driver_parameter()	Driver Settings API
get_driver_settings()	Driver Settings API

B_GET_GEOMETRY	Constants and Defined Types
B_GET_ICON	Constants and Defined Types
B_GET_MEDIA_STATUS	Constants and Defined Types
get_memory_map()	Functions for Drivers & Modules
B_GET_NEXT_OPEN_DEVICE	Constants and Defined Types
B_GET_PARTITION_INFO	Constants and Defined Types
B_GET_READ_STATUS	Constants and Defined Types
B_GET_WRITE_STATUS	Constants and Defined Types
B_GRAPHIC_DRIVER_BASE	Constants and Defined Types

H

has_signals_pending()	Functions for Drivers & Modules
head_count	Constants and Defined Types

I

icon_data	Constants and Defined Types
icon_size	Constants and Defined Types
init_driver()	Writing Drivers
init_hardware()	Writing Drivers
install_io_interrupt_handler()	Functions for Drivers & Modules
Interactions with the Kernel	Device Drivers
io_card_version()	Functions for Drivers & Modules

J

B_JOYSTICK_DRIVER_BASE	Constants and Defined Types

K

The Kernel and the Driver Author	Device Drivers
Kernel Space vs. User Space	Device Drivers
kernel_debugger()	Functions for Drivers & Modules
kprintf()	Functions for Drivers & Modules

L

load_driver_settings()	Driver Settings API
load_driver_symbols()	Functions for Drivers & Modules
B_LOAD_MEDIA	Constants and Defined Types
Loading the Settings	Driver Settings API
lock_memory()	Functions for Drivers & Modules
logical_block_size	Constants and Defined Types

M

malloc()	The area_malloc Module
map_physical_memory()	Functions for Drivers & Modules

B_MIDI_DRIVER_BASE	Constants and Defined Types
Module Functions	The area_malloc Module
Modules	Device Drivers
motherboard_version()	Functions for Drivers & Modules

O

offset	Constants and Defined Types
open_device_iterator	Constants and Defined Types
open_hook()	Writing Drivers

P

panic()	Functions for Drivers & Modules
parse_expression()	Functions for Drivers & Modules
partition	Constants and Defined Types
partition_info	Constants and Defined Types
physical_entry	Functions for Drivers & Modules
platform()	Functions for Drivers & Modules
publish_devices()	Writing Drivers

Q

qent	Functions for Drivers & Modules

R

read_hook()	Writing Drivers
read_only	Constants and Defined Types
Reading a Boolean Parameter	Driver Settings API
Reading a String Parameter	Driver Settings API
Reading All Parameters	Driver Settings API
Reading the Settings	Driver Settings API
readv_hook()	Writing Drivers
realloc()	The area_malloc Module
register_kernel_daemon()	Functions for Drivers & Modules
release_spinlock()	Functions for Drivers & Modules
removable	Constants and Defined Types
remove_debugger_command()	Functions for Drivers & Modules
B_REMOVE_FIXED_DRIVER	Constants and Defined Types
remove_io_interrupt_handler()	Functions for Drivers & Modules
restore_interrupts()	Functions for Drivers & Modules

S

sectors_per_track	Constants and Defined Types
select_hook()	Writing Drivers
send_signal_etc()	Functions for Drivers & Modules
session	Constants and Defined Types
B_SET_BLOCKING_IO	Constants and Defined Types
B_SET_DEVICE_SIZE	Constants and Defined Types
set_dprintf_enabled()	Functions for Drivers & Modules
B_SET_INTERRUPTABLE_IO	Constants and Defined Types

B_SET_NONBLOCKING_IO	Constants and Defined Types
B_SET_PARTITION	Constants and Defined Types
B_SET_UNINTERRUPTABLE_IO	Constants and Defined Types
The Settings File	Driver Settings API
size	Constants and Defined Types
spawn_kernel_thread()	Functions for Drivers & Modules
spin()	Functions for Drivers & Modules
spinlock	Functions for Drivers & Modules
Symbols Drivers Export	Writing Drivers

T

timer	Functions for Drivers & Modules
timer_hook	Functions for Drivers & Modules
Types of Kernel Add-on	Device Drivers

U

uninit_driver()	Writing Drivers
unload_driver_settings()	Driver Settings API
unlock_memory()	Functions for Drivers & Modules
unregister_kernel_daemon()	Functions for Drivers & Modules
Using Modules	anon
Using Modules	Using Modules
Using Modules	Using Modules
Using Posix Calls	Device Drivers

Using Spinlocks	Device Drivers
Using the Driver Settings API	Driver Settings API

W

write_hook()	Writing Drivers
write_once	Constants and Defined Types
writev_hook()	Writing Drivers
Writing Drivers	Writing Drivers
Writing Drivers	Writing Drivers
Writing Modules	anon
Writing Modules	Writing Modules
Writing Modules	Writing Modules

The Be Book **Drivers**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Functions for Drivers & Modules

Summary: *more...*

The kernel exports a number of functions that device drivers can call. The device driver accesses these functions directly in the kernel, not through a library.

Remember when writing a driver that calls one of these functions to link against **_KERNEL_**. This will instruct the loader to dynamically locate the symbols in the current kernel when the driver is loaded.

---

acquire_spinlock() , release_spinlock() , spinlock

Declared in: be/drivers/KernelExport.h

void **acquire_spinlock(**spinlock *_lock_**)**
void **release_spinlock(**spinlock *_lock_**)**
typedef vlong **spinlock**

Spinlocks are mutually exclusive locks that are used to protect sections of code that must execute atomically. Unlike semaphores, spinlocks can be safely used when interrupts are disabled (in fact, you *must* have interrupts disabled).

To create a spinlock, simply declare a `spinlock` variable and initialize it 0:

```
spinlock lock = 0;
```

The functions acquire and release the *lock* spinlock. When you acquire and release a spinlock, you *must* have interrupts disabled; the structure of your code will look like this:

```
cpu_status former = disable_interrupts();
acquire_spinlock(&lock);
/* critical section goes here*/
release_spinlock(&lock);
restore_interrupts(former);
```

The spinlock should be held as briefly as possible, and acquisition must not be nested within the critical section.

Spinlocks are designed for use in a multi-processor system (on a single processor system simply turning off interrupts is enough to guarantee that the critical section will be atomic). Nonetheless, you *can* use spinlocks on a single processor—you don't have to predicate your code based on the number of CPUs in the system.

---

add_debugger_command() see kernel_debugger()

---

add_timer() , cancel_timer() , timer_hook , qent , timer

Declared in: be/drivers/KernelExport.h

```
typedef int32 (*timer_hook)(timer *)
```

```
struct quent = {
    int64   key;
    qent   *next;
    qent   *prev;
    }
```

```
struct timer = {
    qent   entry;
    uint16   flags;
    uint16   cpu;
    timer_hook hook;
    bigtime_t   period;
    }
```

```
status_t add_timer(timer *theTimer, timer_hook hookFunction, bigtime_t period, int32 flags)
```

```
bool cancel_timer(timer_t *theTimer)
```

`add_timer()` installs a new timer interrupt. A timer interrupt causes the specified *hookFunction* to be called when the desired amount of time has passed. On entry, you should pass a pointer to a timer structure in *theTimer*; this will be filled out with data describing the new timer interrupt you've installed. The *flags* argument provides control over how the timer functions, which affects the meaning of the *period* argument as follows:

Constant	Meaning
`B_ONE_SHOT_ABSOLUTE_TIMER`	The timer will fire once at the system time specified by *period*.
`B_ONE_SHOT_RELATIVE_TIMER`	The timer will fire once in approximately *period* microseconds.
`B_PERIODIC_TIMER`	The timer will fire every *period* microseconds, starting in *period* microseconds.

`cancel_timer()` cancels the specified timer. If it's already fired, it returns true; otherwise false is returned. It's guaranteed that once `cancel_timer()` returns, if the timer was in the process of running when `cancel_timer()` was called, the timer function will be finished executing. The only exception to this is if `cancel_timer()` was called from inside a timer handler (in which case trying to wait for the handler to finish running would result in deadlock).

RETURN CODES

- B_OK. The timer was installed (`add_timer()` only).
- `B_BAD_VALUE`. The timer couldn't be installed because the period was invalid (probably because a relative time or period was negative; unfortunately, Be hasn't mastered the

intricacies of installing timers to fire in the past).

---

call_all_cpus()

Declared in: be/drivers/KernelExport.h

void **call_all_cpus**(void (**func*)(void *, int), void **cookie*)

Calls the function specified by *func* on all CPUs. The *cookie* can be anything your needs require.

---

cancel_timer() see add_timer()

---

disable_interrupts() , restore_interrupts() , cpu_status

Declared in: be/drivers/KernelExport.h

typedef ulong **cpu_status**
cpu_status **disable_interrupts**(void)
void **restore_interrupts**(cpu_status *status*)

These functions disable and restore interrupts on the CPU that the caller is currently running on. `disable_interrupts()` returns its previous state (i.e. whether or not interrupts were already disabled). `restore_interrupts()` restores the previous *status* of the CPU, which should be the value that `disable_interrupts()` returned:

```
cpu_status former = disable_interrupts();
...
restore_interrupts(former);
```

As long as the CPU state is properly restored (as shown here), the disable/restore functions can be nested.

See also: **`install_io_interrupt_handler()`**

---

dprintf() , set_dprintf_enabled() , panic()

Declared in: be/drivers/KernelExport.h

void **dprintf**(const char **format*, ...)
bool **set_dprintf_enabled**(bool *enabled*)
void **panic**(const char **format*, ...)

`dprintf()` is a debugging function that has the same syntax and behavior as standard C `printf()`, except that it writes its output to the serial port at a data rate of 19,200 bits per second. The output is sent to **/dev/ports/serial4** on BeBoxes, **/dev/modem** on Macs, and **/dev/ports/serial1** on Intel machines. By default, `dprintf()` is disabled.

`set_dprintf_enabled()` enables `dprintf()` if the *enabled* flag is `true`, and disables it if the flag is `false`. It returns the previous enabled state, thus permitting intelligent nesting:

```
/* Turn on dprintf */
bool former = set_dprintf_enabled(true);
...
/* Now restore it to its previous state. */
set_dprintf_enabled(former);
```

`panic()` is similar to `dprintf()`, except it hangs the computer after printing the message.

---

get_memory_map() , physical_entry

Declared in: be/drivers/KernelExport.h

```
long get_memory_map(const void *address, ulong numBytes,
    physical_entry *table, long numEntries)
typedef struct {   void *address;
    ulong size;
    } physical_entry
```

Returns the physical memory chunks that map to the virtual memory that starts at *address* and extends for *numBytes*. Each chunk of physical memory is returned as a `physical_entry` structure; the series of structures is returned in the *table* array. (which you have to allocate yourself). *numEntries* is the number of elements in the array that you're passing in. As shown in the example, you should lock the memory that you're about to inspect:

```
physical_entry table[count];
lock_memory(addr, extent, 0);
get_memory_map(addr, extent, table, count);
. . .
unlock_memory(someAddress, someNumberOfBytes, 0);
```

The end of the *table* array is indicated by (`size` == 0):

```
long k;
while (table[k].size > 0) {
   /* A legitimate entry */
   if (++k == count) {
      /* Not enough entries */
```

```
        break; }
}
```

If all of the entries have non-zero sizes, then table wasn't big enough; call `get_memory_map()` again with more table entries.

RETURN CODES

The function always returns

- B_OK.

See also: `lock_memory()`, **`start_isa_dma()`**

---

has_signals_pending()

Declared in: be/drivers/KernelExport.h

```
int has_signals_pending(struct thread_rec *thr)
```

Returns a bitmask of the currently pending signals for the current thread. *thr* should always be `NULL`; passing other values will yield meaningless results. `has_signals_pending()` returns 0 if no signals are pending.

---

install_io_interrupt_handler() , remove_io_interrupt_handler()

Declared in: be/drivers/KernelExport.h

```
long install_io_interrupt_handler(long interrupt_number,
    interrupt_handler handler,
    void *data, ulong flags)
```

```
long remove_io_interrupt_handler(long interrupt_number,
    interrupt_handler handler,
    void *data)
```

`install_io_interrupt_handler()` adds the handler *function* to the chain of functions that will be called each time the specified *interrupt* occurs. This function should have the following syntax:

- int32 `handler(`void *_data_`)`

The *data* passed to `install_io_interrupt_handler()` will be passed to the handler function each time it's called. It can be anything that might be of use to the handler, or `NULL`. If the interrupt handler must return one of the following values:

Constant	Meaning
`B_UNHANDLED_INTERRUPT`	The interrupt handler didn't handle the interrupt; the kernel will keep looking for someone to handle it.

B_HANDLED_INTERRUPT	The interrupt handler handled the interrupt. The kernel won't keep looking for a handler to handle it.
B_INVOKE_SCHEDULER	The interrupt handler handled the interrupt. This tells the kernel to invoke the scheduler immediately after the handler returns.

If **B_INVOKE_SCHEDULER** is returned by the interrupt handler, the kernel will immediately invoke the scheduler, to dispatch processor time to tasks that need handling. This is especially useful if your interrupt handler has released a semaphore (see **release_sem_etc()** in the Kernel Kit).

The *flags* parameter is a bitmask of options. The only option currently defined is B_NO_ENABLE_COUNTER. By default, the OS keeps track of the number of functions handling a given interrupt. If this counter changes from 0 to 1, then the system enables the irq for that interrupt. Conversely, if the counter changes from 1 to 0, the system disables the irq. Setting the B_NO_ENABLE_COUNTER flag instructs the OS to ignore the handler for the purpose of enabling and disabling the irq.

install_io_interrupt_handler() returns **B_OK** if successful in installing the handler, and **B_ERROR** if not. An error occurs when either the *interrupt_number* is out of range or there is not enough room left in the interrupt chain to add the handler.

remove_io_interrupt() removes the named *interrupt* from the interrupt chain. It returns B_OK if successful in removing the handler, and B_ERROR if not.

---

io_card_version() see motherboard_version()

---

kernel_debugger() , add_debugger_command() , remove_debugger_command() , load_driver_symbols() , kprintf() , parse_expression()

Declared in: be/drivers/KernelExport.h

void **kernel_debugger**(const char *_string_)
int **add_debugger_command**(char *_name_, int (*_func_)(int, char **), char *_help_)
int **remove_debugger_command**(char *_name_, int (*_func_)(int, char **))
int **load_driver_symbols**(const char *_driverName_)
void **kprintf**(const char *_format_, ...)
ulong **parse_expression**(const char *_string_)

kernel_debugger() drops the calling thread into a debugger that writes its output to the serial port at 19,200 bits per second, just as dprintf() does. This debugger produces *string* as its first message; it's not affected by set_dprintf_enabled().

kernel_debugger() is identical to the **debugger()** function documented in the Kernel Kit, except that it works in the kernel and engages a different debugger. Drivers should use it instead of **debugger()**.

`add_debugger_command()` registers a new command with the kernel debugger. When the user types in the command *name*, the kernel debugger calls func with the remainder of the command line as *argc*/*argv*-style arguments. The help string for the command is set to *help*.

`remove_debugger_command()` removes the specified kernel debugger command.

`load_driver_symbols()` loads symbols from the specified kernel driver into the kernel debugger. **driver_name** is the path-less name of the driver which must be located in one of the standard kernel driver directories. The function returns `B_OK` on success and `B_ERROR` on failure.

`kprintf()` outputs messages to the serial port. It should be used instead of `dprintf()` from new debugger commands because `dprintf()` depends too much upon the state of the kernel to be reliable from within the debugger.

`parse_expression()` takes a C expression and returns the result. It only handles integer arithmetic. The logical and relational operations are accepted. It can also supports variables and assignments. This is useful for strings with multiple expressions, which should be separated with semicolons. Finally, the special variable "." refers to the value from the previous expression. This function is designed to help implement new debugger commands.

See also: `debugger()` in the Kernel Kit

- kprintf() see kernel_debugger()
- load_driver_symbols() see kernel_debugger()

- lock_memory() , unlock_memory()

Declared in: be/drivers/KernelExport.h

long **lock_memory(**void **address*, ulong *numBytes*, ulong *flags***)**
long **unlock_memory(**const void **address*, ulong *numBytes*, ulong *flags***)**

`lock_memory()` makes sure that all the memory beginning at the specified virtual *address* and extending for *numBytes* is resident in RAM, and locks it so that it won't be paged out until `unlock_memory()` is called. It pages in any of the memory that isn't resident at the time it's called. It is typically used in preparation for a DMA transaction.

The *flags* field contains a bitmask of options. Currently, two options, `B_DMA_IO` and `B_READ_DEVICE`, are defined. `B_DMA_IO` should be set if any part of the memory range will be modified by something other than the CPU while it's locked, since that change won't otherwise be noticed by the system and the modified pages may not be written to disk by the virtual memory system. Typically, this sort of change is performed through DMA. `B_READ_DEVICE`, if set, indicates that the caller intends to fill the memory (read *from* the device). If cleared, it indicates the memory will be written to the device and will not be altered.

`unlock_memory()` releases locked memory and should be called with the same flags as passed into the corresponding `lock_memory()` call.

Each of these functions returns **B_OK** if successful and **B_ERROR** if not. The main reason that `lock_memory()` would fail is that you're attempting to lock more memory than can be paged in.

---

map_physical_memory()

Declared in: be/drivers/KernelExport.h

```
area_id map_physical_memory(const char *areaName, void *physicalAddress,
    size_t numBytes, uint32 spec, uint32 protection,
    void **virtualAddress)
```

This function allows you to map the memory in physical memory starting at *physicalAddress* and extending for *numBytes* bytes into your team's address space. The kernel creates an area named *areaName* mapped into the memory address *virtualAddress* and returns its area_id to the caller. *numBytes* must be a multiple of `B_PAGE_SIZE` (4096).

spec must be either `B_ANY_KERNEL_ADDRESS` or `B_ANY_KERNEL_BLOCK_ADDRESS`. If *spec* is `B_ANY_KERNEL_ADDRESS`, the memory will begin at an arbitrary location in the kernel address space. If *spec* is `B_ANY_KERNEL_BLOCK_ADDRESS`, then the memory will be mapped into a memory location aligned on a multiple of `B_PAGE_SIZE`.

protection is a bitmask consisting of the fields `B_READ_AREA` and `B_WRITE_AREA`, as discussed in **`create_area()`**.

`create_area()` returns an area_id for the newly-created memory if successful or an error code on failure. The error codes are the same as those for **`create_area()`**.

See also: **`create_area()`**

---

motherboard_version() , io_card_version()

Declared in: be/drivers/KernelExport.h

```
long motherboard_version(void)
long io_card_version(void)
```

These functions return the current versions of the motherboard and of the I/O card. These functions are only available on PowerPC-based systems (they're intended for use on the BeBox).

---

panic() see dprintf()

parse_expression() see kernel_debugger()

---

platform()

Declared in: be/drivers/KernelExport.h

platform_type **platform**(void)

Returns the current platform, as defined in **<kernel/OS.h>**.

---

register_kernel_daemon() , unregister_kernel_daemon()

Declared in: be/drivers/KernelExport.h

int **register_kernel_daemon**(void (**func*)(void *, int), void **arg*, int *freq*)
int **unregister_kernel_daemon**(void (**func*)(void *, int), void **arg*)

Adds or removes daemons from the kernel. A kernel daemon function is executed approximately once every *freq*/10 seconds. The kernel calls *func* with the arguments *arg* and an iteration value that increases by *freq* on successive calls to the daemon function.

- release_spinlock() see acquire_spinlock()
- remove_io_interrupt_handler() see install_io_interrupt_handler()
- restore_interrupts() see disable_interrupts()
- set_dprintf_enabled() see dprintf()

---

send_signal_etc()

Declared in: be/drivers/KernelExport.h

int **send_signal_etc**(pid_t *thid*, uint *sig*, uint32 *flags*)

This function is a counterpart to **`send_signal()`** in the Posix layer, which is not exported for drivers.

thid is the thread_id of the thread the signal should be sent to, and sig is the signal type to send, just like in **`send_signal()`**. The *flags* argument can be used to specify flags to control the function:

Constant	Meaning
`B_CHECK_PERMISSION`	The signal will only be sent if the destination thread's uid and euid are the same as the caller's.
`B_DO_NOT_RESCHEDULE`	The kernel won't call the scheduler after sending the signal. You should specify this flag when calling `send_signal_etc()` from an interrupt handler.

RETURN CODES

- B_OK. The signal was sent.
- **B_BAD_VALUE**. The signal type is invalid.
- **B_BAD_THREAD_ID**. The thread ID is invalid.
- **B_NOT_ALLOWED**. The permission check failed (if **B_CHECK_PERMISSION** was specified).

---

spawn_kernel_thread()

Declared in: be/drivers/KernelExport.h

```
thread_id spawn_kernel_thread(thread_entry func, const char *name,
    long priority, void *data)
```

This function is a counterpart to **spawn_thread()** in the Kernel Kit, which is not exported for drivers. It has the same syntax as the Kernel Kit function, but is able to spawn threads in the kernel's memory space.

See also: **spawn_thread()** in the Kernel Kit

---

spin()

Declared in: be/drivers/KernelExport.h

```
void spin(bigtime_t microseconds)
```

Executes a delay loop lasting at least the specified number of *microseconds*. It could last longer, due to rounding errors, interrupts, and context switches.

---

un see lock_memory()
un see register_kernel_daemon()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Functions for Drivers & Modules Function Summary

Function Summary

Function	Description
acquire_spinlock()	Acquires a spinlock
add_debugger_command()	Adds a new command to the kernel debugger
add_timer()	Installs a timer interrupt
call_all_cpus()	Calls a function on all processors
cancel_timer()	Removes a timer interrupt
disable_interrupts()	Disables interrupts
dprintf()	Prints a debugging message to the debug serial port
get_memory_map()	Returns the physical memory chunk corresponding to a given virtual memory address
has_signals_pending()	Indicates what signals are currently pending for a given thread
install_io_interrupt_handler()	Installs an I/O interrupt handler
io_card_version()	Returns the version of the I/O card
kernel_debugger()	Drops the thread into the kernel debugger

kprintf()	Prints a message to the serial port (for use in debugger commands)
load_driver_symbols()	Loads the specified driver's symbols into the debugger
lock_memory()	Locks the specified range of memory into physical memory
map_physical_memory()	Maps the specified physical memory into your team's address space
motherboard_version()	Returns the version of the motherboard
panic()	Prints a debugging message then halts the system
parse_expression()	Parses a C expression and returns the result
platform()	Returns the current platform
register_kernel_daemon()	Adds a daemon to the kernel
release_spinlock()	Releases a spinlock
remove_debugger_command()	Removes a command from the kernel debugger
remove_io_interrupt_handler()	Removes an I/O interrupt handler
restore_interrupts()	Restores the previous interrupt state

send_signal_etc()	Sends a signal to a specified thread
set_dprintf_enabled()	Turns on or off debug output
spawn_kernel_thread()	Spawns a new thread in the kernel's address space
spin()	Delays for at least a given number of microseconds.
unlock_memory()	Unlocks the specified range of memory
unregister_kernel_daemon()	Removes a daemon from the kernel

The Be Book **Drivers** **Drivers Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Areas

Declared in: be/kernel/OS.h
Library: libroot.so
Summary: *more...*

An area is a chunk of virtual memory that can be shared between threads (possibly in different teams). If your application needs to allocate large chunks of memory, or wants to share lots of data with another application, you should consider using an area.

For more on area concepts, see "Areas Concepts".

For examples of creating and sharing areas, see "Area Examples".

Area Functions

area_for()

```
area_id area_for(void *addr)
```

Returns the area that contains the given address (within your own team's address space). The argument needn't be the starting address of an area, nor must it start on a page boundary: If the address lies anywhere within one of your application's areas, the ID of that area is returned.

Since the address is taken to be in the local address space, the area that's returned will also be local—it will have been created or cloned by your application.

RETURN CODES

- `B_ERROR`. The address doesn't lie within an area.

See also: `find_area()`

clone_area()

```
area_id clone_area(const char *clone_name,
    void **clone_addr,
    uint32 clone_addr_spec,
    uint32 clone_protection,
    area_id source_area)
```

Creates a new area (the *clone* area) that maps to the same physical memory as an existing area (the *source* area).

- *clone_name* is the name that you wish to assign to the clone area. Area names are, at most, `B_OS_NAME_LENGTH` characters long.
- *clone_addr* points to a value that gives the address at which you want the clone area to start; the pointed-to value must be a multiple of `B_PAGE_SIZE` (4096). The function sets the value pointed to by *clone_addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **clone_addr* depends on the value of *clone_addr_spec*, as explained next.
- *clone_addr_spec* is one of four constants that describes how *clone_addr* is to be interpreted. The first three constants, **B_EXACT_ADDRESS**, **B_BASE_ADDRESS**, and **B_ANY_ADDRESS**, have meanings as explained under `create_area()`.

The fourth constant, **B_CLONE_ADDRESS**, specifies that the address of the cloned area should be the same as the address of the source area. Cloning the address is convenient if you have two

(or more) applications that want to pass pointers to each other—by using cloned addresses, the applications won't have to offset the pointers that they receive. For both the **B_ANY_ADDRESS** and **B_CLONE_ADDRESS** specifications, the value that's pointed to by the *clone_addr* argument is ignored.

- *clone_protection* is one or both of **B_READ_AREA** and **B_WRITE_AREA**. These have the same meaning as in **create_area()**; keep in mind, as described there, that a cloned area can have a protection that's different from that of its source.
- *source_area* is the **area_id** of the area that you wish to clone. You usually supply this value by passing an area name to the **find_area()** function.

The cloned area inherits the source area's locking scheme.

Usually, the source area and clone area are in two different applications. It's possible to clone an area from a source that's in the same application, but there's not much reason to do so unless you want the areas to have different protections.

If `clone_area()` clone is successful, the clone's **area_id** is returned. Otherwise, it returns a descriptive error code, listed below.

RETURN CODES

- **B_BAD_VALUE**. Bad argument value; you passed an unrecognized constant for *addr_spec* or *lock*, the *addr* value isn't a multiple of **B_PAGE_SIZE**, you set *addr_spec* to **B_EXACT_ADDRESS** or **B_CLONE_ADDRESS** but the address request couldn't be fulfilled, or *source_area* doesn't identify an existing area.
- **B_NO_MEMORY**. Not enough memory to allocate the system structures that support this area (unlikely).
- **B_ERROR**. Some other system error prevented the area from being created.

See also: **create_area()**

---

create_area()

```
area_id create_area(const char *name,
    void **addr,
    uint32 addr_spec,
    uint32 size,
    uint32 lock,
    uint32 protection)
```

Creates a new area and returns its **area_id**.

- *name* is the name that you wish to assign to the area. It needn't be unique. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.

- *addr* points to the address at which you want the area to start. The value of **addr* must signify a page boundary; in other words, it must be an integer multiple of **B_PAGE_SIZE** (4096). Note that this is a pointer to a pointer: **addr*—not *addr*—should be set to the desired address; you then pass the address of *addr* as the argument, as shown below:

```
/* Set the address to a page boundary. */
char *addr = (char *)(B_PAGE_SIZE * 100);

/* Pass the address of addr as the second argument. */
create_area( "my area", &addr, ...);
```

The function sets the value of **addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **addr* depends on the value of *addr_spec*, as explained next.

- *addr_spec* is a constant that tells the function how the **addr* value should be applied. There are three useful address specification constants, and one that doesn't apply here:

Constant	Meaning
B_EXACT_ADDRESS	You want the value of **addr* to be taken literally and strictly. If the area can't be allocated at that location, the function fails.
B_BASE_ADDRESS	The area can start at a location equal to or greater than **addr*.
B_ANY_ADDRESS	The starting address is determined by the system. In this case, the value that's pointed to by *addr* is ignored (going into the function).
B_ANY_KERNEL_ADDRESS	The starting address is determined by the system, and the new area will belong to the kernel's team; it won't be deleted when the application quits. In this case, the value that's pointed to by *addr* is ignored (going into the function).
B_CLONE_ADDRESS	This is only meaningful to the **clone_area()** function.

- *size* is the size, in bytes, of the area. The size must be an integer multiple of **B_PAGE_SIZE** (4096). The upper limit of *size* depends on the available swap space (or RAM, if the area is to be locked).

- *lock* describes how the physical memory should be treated with regard to swapping. There are four locking constants:

Constant	Meaning
B_FULL_LOCK	The area's memory is locked into RAM when the area is created, and won't be swapped out.
B_CONTIGUOUS	Not only is the area's memory locked into RAM, it's also guaranteed to be contiguous. This is particularly—and perhaps exclusively—useful to designers of certain types of device drivers.
B_LAZY_LOCK	Allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.
B_NO_LOCK	Pages are never locked, they're swapped in and out as needed.

B_LOMEM	This is a special constant that's used for for areas that need to be locked, contiguous, and that fit within the first 16MB of physical memory. The folks that need this constant know who they are.

- *protection* is a mask that describes whether the memory can be written and read. You form the mask by adding the constants **B_READ_AREA** (the area can be read) and **B_WRITE_AREA** (it can be written). The protection you describe applies only to this area. If your area is cloned, the clone can specify a different protection.

If `create_area()` is successful, the new **area_id** number is returned. If it's unsuccessful, one of the following error constants is returned.

RETURN CODES

- **B_BAD_VALUE**. Bad argument value. You passed an unrecognized constant for *addr_spec* or *lock*, the *addr* or *size* value isn't a multiple of **B_PAGE_SIZE**, or you set *addr_spec* to **B_EXACT_ADDRESS** but the address request couldn't be fulfilled.
- **B_NO_MEMORY**. Not enough memory to allocate the system structures that support this area (unlikely), not enough physical memory to support a locked area, or not enough swap space to allocate virtual memory (in other words, *size* is too big.)
- **B_ERROR**. Some other system error prevented the area from being created.

See also: `clone_area()`

---

delete_area()

status_t **delete_area(**area_id *area***)**

Deletes the designated area. If no one other area maps to the physical memory that this area represents, the memory is freed. After being deleted, the *area* value is invalid as an area identifier.

Currently, anybody can delete any area—the act isn't denied if, for example, the **area_id** argument was created by another application. This freedom will be rescinded in a later release. Until then, try to avoid deleting other application's areas.

RETURN CODES

- **B_OK**. The area was deleted; *area* is now invalid.
- **B_ERROR**. *area* doesn't designate an actual area.

---

find_area()

area_id **find_area(**const char *_name_**)**

Returns an area that has a name that matches the argument. Area names needn't be unique—successive calls to this function with the same argument value may not return the same **area_id**.

What you do with the area you've found depends on where it came from:

- If you're finding an area that your own application created or cloned, you can use the returned ID directly.
- If the area was created or cloned by some other application, you should immediately clone the area (unless you're doing something truly innocuous, such as simply examining the area's info structure).

RETURN CODES

- **B_NAME_NOT_FOUND**. The argument doesn't identify an existing area.

See also: **area_for()**

---

get_area_info() , get_next_area_info() , area_info

status_t **get_area_info(**area_id *area*, area_info **info***)*
status_t **get_next_area_info(**team_id *team*, int32 **cookie*, area_info **info***)*
struct {} **area_info**

Copies information about a particular area into the `area_info` structure designated by *info*. The first version of the function designates the area directly, by **area_id**.

The `get_next_area_info()` version lets you step through the list of a team's areas through iterated calls on the function. The *team* argument identifies the team you want to look at; a *team* value of 0 means the team of the calling thread. The *cookie* argument is a placemark; you set it to 0 on your first call, and let the function do the rest. The function returns **B_BAD_VALUE** when there are no more areas to visit:

```
/* Get the area_info for every area in this team. */
area_info info;
int32 cookie = 0;

while (get_next_area_info(0, &cookie, &info) == B_OK)
   ...
```

The `area_info` structure is:

```
typedef struct area_info {
      area_id area;
      char name[B_OS_NAME_LENGTH];
      size_t size;
```

```
    uint32 lock;
    uint32 protection;
    team_id team;
    size_t ram_size;
    uint32 copy_count;
    uint32 in_count;
    uint32 out_count;
    void *address;
} area_info;
```

The fields are:

- `area` is the **`area_id`** that identifies the area.
- `name` is the name that was assigned to the area when it was created or cloned.
- `size` is the (virtual) size of the area, in bytes.
- `lock` is a constant that represents the area's locking scheme. This will be one of **`B_FULL_LOCK`**, **`B_CONTIGUOUS`**, **`B_LAZY_LOCK`**, **`B_NO_LOCK`**, or **`B_LOMEM`**.
- `protection` specifies whether the area's memory can be read or written. It's a combination of **`B_READ_AREA`** and **`B_WRITE_AREA`**.
- `team` is the **`team_id`** of the team that created or cloned this area.
- `address` is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the team that created (or cloned) the area.

The final four fields give information about the area that's useful in diagnosing system use. The fields are particularly valuable if you're hunting for memory leaks:

- `ram_size` gives the amount of the area, in bytes, that's currently swapped in.
- `copy_count` is a "copy-on-write" count that can be ignored—it doesn't apply to the areas that you create. The system can create copy-on-write areas (it does so when it loads the data section of an executable, for example), but you can't.
- `in_count` is a count of the total number of times any of the pages in the area have been swapped in.
- `out_count` is a count of the total number of times any of the pages in the area have been swapped out.

RETURN CODES

- **`B_OK`**. The area was found; *info* contains valid information.
- **`B_BAD_VALUE`**. *area* doesn't identify an existing area, *team* doesn't identify an existing team, or there are no more areas to visit.

---

resize_area()

```
status_t resize_area(area_id area, size_t new_size)
```

Sets the size of the designated area to *new_size*, measured in bytes. The *new_size* argument must be a multiple of `B_PAGE_SIZE` (4096).

Size modifications affect the end of the area's existing memory allocation: If you're increasing the size of the area, the new memory is added to the end of area; if you're shrinking the area, end pages are released and freed. In neither case does the area's starting address change, nor is existing data modified (except, of course, for data that's lost due to shrinkage).

Resizing affects all areas that refer to this areas physical memory. For example, if B is a clone of A, and you resize A, B will be automatically resized (if possible).

RETURN CODES

- `B_OK`. The area was successfully resized.
- `B_BAD_VALUE`. *area* doesn't signify a valid area, or *new_size* isn't a multiple of `B_PAGE_SIZE`.
- `B_NO_MEMORY`. Not enough memory to support the new portion of the area. This should only happen if you're increasing the size of the area.
- `B_ERROR`. Some other system error prevented the area from being created.

---

set_area_protection()

```
status_t set_area_protection(area_id area, uint32 new_protection)
```

Sets the given area's read and write protection. The *new_protection* argument is a mask that specifies one or both of the values **`B_READ_AREA`** and **`B_WRITE_AREA`**. The former means that the area can be read; the latter, that it can be written to. An area's protection only applies to access to the underlying memory through that specific area. Different area clones that refer to the same memory may have different protections.

RETURN CODES

- `B_OK`. The protection was changed.
- `B_BAD_VALUE`. *area* doesn't identify a valid area.

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Areas Function Summary

Function Summary

Function	Description
area_for()	Returns area containing a given address
clone_area()	Creates a new area mapping to the same memory as the source
create_area()	Creates a new area
delete_area()	Deletes a specified area
find_area()	Returns the area with the given name
get_area_info()	Returns information about a specific area
get_next_area_info()	Enumerates information about all areas for a given team
resize_area()	Sets the size of a specified area
set_area_protection()	Sets the read/write protection of a given area

---

Copyright © 2000 Be, Inc. All rights reserved..

Areas Concepts

An area is a chunk of virtual memory. As such, it has all the expected properties of virtual memory: It has a starting address, a size, the addresses it comprises are contiguous, and it maps to (possibly non-contiguous) physical memory. The features that an area provides that you *don't* get with "standard" memory are these:

- **Areas can be shared**. Different areas can refer to the same physical memory. Put another way, different virtual memory addresses can map to the same physical locations. Furthermore, the different areas needn't belong to the same application. By creating and "cloning" areas, applications can easily share the same data.
- **Areas can be locked into RAM**. You can specify that the area's physical memory be locked into RAM when it's created, locked on a page-by-page basis as pages are swapped in, or that it be swapped in and out as needed.
- **Areas can be read- and write-protected**.
- **Areas are page-aligned**. Areas always start on a page boundary, and are allocated in integer multiples of the size of a page. (A page is 4096 bytes, as represented by the `B_PAGE_SIZE` constant.)
- **You can specify the starting address of the area's virtual memory**. The specification can require that the

area start precisely at a certain address, anywhere above a certain address, or anywhere at all.

Because areas are large—one page, minimum—you don't create them arbitrarily. The two most compelling reasons to create an area are the two first points listed above: To share data among different applications, and to lock memory into RAM.

In all particulars (but one) you treat the memory that an area gives you exactly as you would treat any allocated memory: You can read and write it through pointer manipulation, or through standard functions such as **`memcpy()`** and **`strcpy()`**. The one difference is between areas and malloc'd memory is...

- You never **`free()`** the memory that an area allocates for you. If you want to get rid of an area, use the **`delete_area()`** function, instead.

Area IDs and Area Names

Each area that you create is tagged with an **`area_id`** number:

- An **`area_id`** number is a positive integer that's global and unique within the scope of the computer. They're not unique across the network, nor are they persistent across boots.

- The **`area_id`** numbers are generated and assigned automatically by the **`create_area()`** and **`clone_area()`** functions. The other area functions operate on these **`area_id`** numbers (they're required as arguments).

- Although they are global, **area_id** numbers have little meaning outside of the address space (application) in which they were created.
- Once assigned, the **area_id** number doesn't change; the number is invalidated when **delete_area()** is called or when the application (team) that created it dies.
- Don't worry about recycled **area_id** numbers. When an area is deleted, it's **area_id** goes with it. (**area_id** values *are* recycled, but the turnover is at 2^31.)

Areas can also be (loosely) identified by name:

- When you create an area (through **create_area()** or **clone_area()**), you get to name it.
- Area names are not unique—any number of areas can be assigned the same name.
- To look up an area by name, use the **FindArea()** function.

Sharing an Area Between Applications

For multiple applications to share a common area, one of the applications has to *create* the area, and the other applications *clone* the area. You clone an area by calling **clone_area()**. The function takes, as its last argument, the **area_id** of the source area and returns a new (unique) **area_id** number. All further references to the cloned area (in the cloning application) must be based on the area_id that's returned by **clone_area()**.

So how does a cloner find a source **area_id** in the first place?

- The source application can pass the "original" **area_id** number to the cloners.
- The cloners can find the area by name, by calling **find_area()**.

Keep in mind that area names are not forced to be unique, so the **find_area()** method has some amount of uncertainty. But this can be minimized through clever name creation.

Cloned Memory

The physical memory that lies beneath an area is never implicitly copied—for example, the area mechanism doesn't perform a "copy-on-write." If two areas refer to the same memory because of cloning, a data modification that's affected through one area will be seen by the other area.

Locking an Area

When you're working with moderately large amounts of data, it's often the case that you would prefer that the data remain in RAM, even if the rest of your application needs to be swapped out. An argument to **create_area()** lets you declare, through the use of one of the following constants, the locking scheme that you wish to apply to your area:

Constant	Meaning
B_FULL_LOCK	The area's memory is locked into RAM when the area is created, and won't be swapped out.
B_CONTIGUOUS	Not only is the area's memory locked into RAM, it's also guaranteed to be contiguous.

	This is particularly—and perhaps exclusively—useful to designers of certain types of device drivers.
B_LAZY_LOCK	Allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.
B_NO_LOCK	Pages are never locked, they're swapped in and out as needed.
B_LOMEM	This is a special constant that's used for for areas that need to be locked, contiguous, and that fit within the first 16MB of physical memory. The folks that need this constant know who they are.

Keep in mind that locking an area essentially reduces the amount of RAM that can be used by other applications, and so increases the likelihood of swapping. So you shouldn't lock simply because you're greedy. But if the area that you're locking is going to be shared among some number of other applications, or if you're writing a real-time application that processes large chunks of data, then locking can be a justifiable excess.

The locking scheme is set by the **create_area()** function and is thereafter immutable. You can't re-declare the lock when you clone an area.

Area Info

Ultimately, you use an area for the virtual memory that it represents: You create an area because you want some

memory to which you can write and from which you can read data. These acts are performed in the usual manner, through references to specific addresses. Setting a pointer to a location within the area, and checking that you haven't exceeded the area's memory bounds as you increment the pointer (while reading or writing) are your own responsibility. To do this properly, you need to know the area's starting address and its extent:

- An area's starting address is maintained as the `address` field in its **`area_info`** structure; you retrieve the **`area_info`** for a particular area through the **`get_area_info()`** function.
- The size of the area (in bytes) is given as the `size` field of its **`area_info`** structure.

An important point, with regard to **`area_info`**, is that the `address` field is only valid for the application that created or cloned the area (in other words, the application that created the **`area_id`** that was passed to **`get_area_info()`**). Although the memory that underlies an area is global, the address that you get from an **`area_info`** structure refers to a specific address space.

If there's any question about whether a particular **`area_id`** is "local" or "foreign," you can compare the `area_info.team` field to your thread's team.

Deleting an Area

When your application quits, the areas (the **`area_id`** numbers) that it created through **`create_area()`** or

clone_area() are automatically rendered invalid. The memory underlying these areas, however, isn't necessarily freed. An area's memory is freed only when (and as soon as) there are no more areas that refer to it.

You can force the invalidation of an **area_id** by passing it to the **delete_area()** function. Again, the underlying memory is only freed if yours is the last area to refer to the memory.

Deleting an area, whether explicitly through **delete_area()**, or because your application quit, never affects the status of other areas that were cloned from it.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Area Examples

Example 1: Creating and Writing into an Area

Example 2: Reading a File into an Area

Example 3: Accessing a Designated Area

Example 4: Cloning and Sharing an Area

Example 5: Cloning Addresses

Example 1: Creating and Writing into an Area

As a simple example of area creation and usage, here we create a ten page area and fill half of it (with nonsense) by bumping a pointer:

```
area_id my_area;
char *area_addr, *ptr;

/* Create an area. */
my_area = create_area("my area", /* name you give to the area */
      (void *)&area_addr, /* returns the starting addr */
      B_ANY_ADDRESS, /* area can start anywhere */
      B_PAGE_SIZE*10, /* size in bytes */
      B_NO_LOCK, /* Lock in RAM? No. */
      B_READ_AREA | B_WRITE_AREA); /* permissions */

/* check for errors */
if (my_area < 0) {
      printf("Something bad happenedn");
      return;
}

/* Set ptr to the beginning of the area. */
ptr = area_addr;

/* Fill half the area (with random-ish data). */
for (int i; i < B_PAGE_SIZE*5; i++)
   *ptr++ = system_time()%256;
```

You can also **`memcpy()`** and **`strcpy()`** into the area:

```
/* Copy the first half of the area into the second half. */
memcpy(ptr, area_addr, B_PAGE_SIZE*5);

/* Overwrite the beginning of the area. */
strcpy(area_addr, "Hey, look where I am.");
```

When we're all done, we delete the area:

```
delete_area(my_area);
```

Example 2: Reading a File into an Area

Here's a function that finds a file, opens it (implicit in the BFile constructor), and copies its contents into RAM:

```
#include <File.h>

area_id file_area;

status_t file_reader(const char *pathname)
{
   status_t err;
   char *area_addr;

   BFile file(pathname, B_READ_ONLY);
   if ((err=file.InitCheck()) != B_OK) {
      printf("%s: Can't find or open.n", pathname);
      return err;
   }

   err = file.GetSize(&file_size);
   if (err != B_OK || file_size == 0) {
      printf("%s: Disappeared? Empty?n", pathname);
      return err;
   }

   /* Round the size up to the nearest page. */
   file_size = (((file_size-1)%B_PAGE_SIZE)+1)*B_PAGE_SIZE;

   /* Make sure the size won't overflow a size_t spec. */
   if (file_size >= ((1<<32)-1) ) {
      printf("%s: What'd you do? Read Montana?n");
      return B_NO_MEMORY;
   }
   file_area = create_area("File area", (void *)&area_addr,
      B_ANY_ADDRESS, file_size, B_FULL_LOCK,
      B_READ_AREA | B_WRITE_AREA);

   /* Check create_area() errors, as in the last example. */
   ...

   /* Read the file; delete the area if there's an error. */
   if ((err=file.Read(area_addr, file_size)) < B_OK) {
      printf("%s: File read error.n");
      delete_area(file_area);
      return err;
```

```
    }

    /* The file is automatically closed when the stack-based
    * BFile is destroyed.
    */
    return B_OK;
}
```

Example 3: Accessing a Designated Area

In the previous example, a local variable (`area_addr`) was used to capture the starting address of the newly-created area. If some other function wants to access the area, it must "re-find" the starting address (and the length of the area, for boundary checking). To do this, you call **`get_area_info()`**.

In the following example, an area is passed in by name; the function, which will write its argument buffer to the area, calls **`get_area_info()`** to determine the start and extent of the area, and also to make sure that the area is part of this team. If the area was created by some other team, the function could still write to it, but it would have to clone the area first (cloning is demonstrated in the next example).

```
status_t write_to_area(const char *area_name,
               const void *buf,
               size_t len)
{
   area_id area;
   area_info ai;
   thread_id thread;
   thread_info ti;
   status_t err;

   if (!area_name)
      return B_BAD_VALUE;

   area = find_area(area_name);

   /* Did we find it? */
   if (area < B_OK) {
      printf("Couldn't find area %s.n", area_name);
      return err;
   }

   /* Get the info. */
   err = get_area_info(area, &ai);

   if (err < B_OK) {
      printf("Couldn't get area info.n");
      return err;
   }

   /* Get the team of the calling thread; to do this, we have
   * to look in the thread_info structure.
   */
   err = get_thread_info(find_thread(NULL), &ti);
```

```
   if (err < B_OK) {
      printf("Couldn't get thread info.n");
      return err;
   }

   /* Compare this team to the area's team. */
   if (ai.team != ti.team)
      printf("Foreign area.n");
      return B_NOT_ALLOWED;
   }
   /* Make sure we're not going to overflow the area,
   * and make sure this area can be written to.
   */
   if (len > ai.size) {
      printf("Buffer bigger than area.n");
      return B_BAD_VALUE;
   }
   if (!(ai.protection & B_WRITE_AREA)) {
      printf("Can't write to this area.n");
      return B_NOT_ALLOWED;
   }

   /* Now we can write. */
   memcpy(ai.address, buf, len);
   return B_OK;
}
```

It's important that you only write to areas that were created or cloned within the calling team. The starting address of a "foreign" area is usually meaningless within your own address space.

You don't *have* to check the area's protectection before writing to it (or reading from it). The memory-accessing fucntions (`memcpy()`, in this example) will segfault if an invalid read or write is requested.

Example 4: Cloning and Sharing an Area

In the following example, a server and a client are set up to share a common area. Here's the server:

```
/* Server side */
class AServer
{
   status_t make_shared_area(size_t size);
   area_id the_area;
   char *area_addr;
};

status_t AServer::make_shared_area(size_t size)
{
   /* The size must be rounded to a page. */
   size = ((size % B_PAGE_SIZE)+1) * B_PAGE_SIZE;
   the_area = create_area("server area", (void *)&area_addr
            B_ANY_ADDRESS, size, B_NO_LOCK,
            B_READ_AREA|B_WRITE_AREA);

   if (the_area < B_OK) {
      printf("Couldn't create server arean");
      return the_area;

   return B_OK;
}
```

And here's the client:

```
/* Client side */
class AClient
{
   status_t make_shared_clone();
   area_id the_area;
   char *area_addr;
};

status_t AClient::make_shared_clone()
{
   area_id src_area;

   src_area = find_area("server area");
   if (src_area < B_ERROR) {
      printf("Couldn't find server area.n");
      return src_area;
```

```
   }
   the_area = clone_area("client area",
               (void *)&area_addr,
                  B_ANY_ADDRESS,
               B_READ_AREA | B_WRITE_AREA,
               src_area);

   if (the_area < B_OK)
      printf("Couldn't create clone arean");
      return the_area;
   }
   return B_OK;
}
```

Notice that the area creator (the server in the example) doesn't have to designate the created area as sharable. All areas are candidates for cloning.

After it creates the cloned area, the client's **area_id** value (`AClient::the_area`) will be different from the server's (`AServer::the_area`). Even though **area_id** numbers are global, the client should only refer to the server's **area_id** number in order to clone it. After the clone, the client talks to the area through its own **area_id** (the value passed backed by **clone_area()**).

Example 5: Cloning Addresses

It's sometimes useful for shared areas (in other words, a "source" and a clone) to begin at the same starting address. For example, if a client's clone area starts at the same address as the server's original area, then the client and server can pass area-accessing pointers back and forth without having to translate the addresses. Here we modify the previous example to do this:

```
status_t AClient::make_shared_clone()
{
   area_id src_area;

   src_area = find_area("server area");

   if (src_area < B_ERROR) {
      printf("Couldn't find server area.n");
      return B_BAD_VALUE;
   }

   /* This time, we specify the address that we want the
   * clone to start at. The B_CLONE_ADDRESS constant
   * does this for us.
   */
   area_addr = src_info.address;
   the_area = clone_area("client area",
               (void *)&area_addr,
                  B_CLONE_ADDRESS,
               B_READ_AREA | B_WRITE_AREA,
               src_area);

   if (the_area < B_OK)
      printf("Couldn't create clone arean");
      return the_area;
   }
   return B_OK;
}
```

Of course, demanding that an area begin at a specific address can be too restrictive; if any of the memory within [area_addr, area_addr + src_info.size] is already allocated, the clone will fail.

The Be Book The Kernel Kit The Kernel Kit Index

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Constants and Defined Types

Declared in: be/drivers/Drivers.h
more...

This section covers constants and types defined for use by kernel drivers and modules.

Constants

Current Driver API Version

```
#define B_CUR_DRIVER_API_VERSION N
```

The **B_CUR_DRIVER_API_VERSION** constant indicates what version of the driver API your driver will be built to.

See also: "Symbols Drivers Export"

Driver Control Opcodes

Constant	Meaning
B_GET_DEVICE_SIZE	Returns a **size_t** indicating the device size in bytes.
B_SET_DEVICE_SIZE	Sets the device size to the value pointed to by *data*.
B_SET_NONBLOCKING_IO	Sets the device to use nonblocking I/O.
B_SET_BLOCKING_IO	Sets the device to use blocking I/O.
B_GET_READ_STATUS	Returns **true** if the device can read without blocking, otherwise **false**.
B_GET_WRITE_STATUS	Returns **true** if the device can write without blocking, otherwise **false**.
B_GET_GEOMETRY	Fills out the specified **device_geometry** structure to describe the device.
B_GET_DRIVER_FOR_DEVICE	Returns the path of the driver executable handling the device.
B_GET_PARTITION_INFO	Returns a **partition_info** structure for the device.
B_SET_PARTITION	Creates a user-defined partition. *data* points to a **partition_info** structure.
B_FORMAT_DEVICE	Formats the device. *data* should point to a boolean value: If **true**, the device is formatted low-level. If it's **false**, the formatting is <<??>>
B_EJECT_DEVICE	Ejects the device.
B_GET_ICON	Fills out the specified **device_icon** structure to describe the device's icon.
B_GET_BIOS_GEOMETRY	Fills out a **device_geometry** structure to describe the device as the BIOS sees it.
B_GET_MEDIA_STATUS	Gets the status of the media in the device by placing a **status_t** at the location pointed to by *data*. (See the list below.)
B_LOAD_MEDIA	Loads the media.
B_GET_BIOS_DRIVE_ID	Returns the BIOS ID for the device.

B_SET_UNINTERRUPTABLE_IO	Prevents control-C from interrupting I/O.
B_SET_INTERRUPTABLE_IO	Allows control-C to interrupt I/O.
B_FLUSH_DRIVE_CACHE	Flushes the drive's cache.
B_GET_NEXT_OPEN_DEVICE	Iterates through open devices; *data* points to an **open_device_iterator**.
B_ADD_FIXED_DRIVER	For internal use only.
B_REMOVE_FIXED_DRIVER	For internal use only.
B_AUDIO_DRIVER_BASE	Base for codes in **audio_driver.h**.
B_MIDI_DRIVER_BASE	Base for codes in **midi_driver.h**.
B_JOYSTICK_DRIVER_BASE	Base for codes in **joystick.h**.
B_GRAPHIC_DRIVER_BASE	Base for codes in **graphic_driver.h**.
B_DEVICE_OP_CODES_END	End of Be-defined control IDs.

B_GET_MEDIA_STATUS returns one of these values in *data*:

- **B_NO_ERROR**. The media is ready.
- **B_DEV_NO_MEDIA**. No media in the device
- **B_DEV_NOT_READY**. The device isn't ready.
- **B_DEV_MEDIA_CHANGED**. The media changed since the time that the device was opened, or since the time of the last **B_GET_MEDIA_STATUS** request.
- **B_DEV_MEDIA_CHANGE_REQUEST**. The user wants to remove the media.
- **B_DEV_DOOR_OPEN**. The drive "door" is open.

Defined Types

device_geometry

```
typedef struct {
    uint32 bytes_per_sector;
    uint32 sectors_per_track;
    uint32 cylinder_count;
    uint32 head_count;
    uchar device_type;
    bool removable;
    bool read_only;
    bool write_once;
    } device_geometry
```

The `device_geometry` structure is returned by the `B_GET_GEOMETRY` driver control function. Its fields are:

- **`bytes_per_sector`** indicates how many bytes each sector of the disk contains.
- **`sectors_per_track`** indicates how many sectors each disk track contains.
- **`cylinder_count`** indicates the number of cylinders the disk contains.
- **`head_count`** indicates how many heads the disk has.
- **`device_type`** specifies the type of device; there's a list of device type definitions below.
- **`removable`** is **`true`** if the device's media can be removed from the drive, and is **`false`** otherwise.
- **`read_only`** is **`true`** if the media is read-only (such as CD-ROM), or **`false`** if the media can be both read from and written .
- **`write_once`** is **`true`** if the media can only be written to once (such as CD-recordable), or **`false`** if there's no limit to the number of times the media can be written to.

If you need to compute the total size of the device in bytes, you can obtain this figure using the following simple formula:

```
disk_size = geometry.cylinder_count * geometry.sectors_per_track * geometry.head_count *
geometry.bytes_per_sector;
```

The device type returned in **`device_type`** is one of:

- **`B_DISK`**. Hard disk, floppy disk, etc.
- **`B_TAPE`**. Tape drive.

- **B_PRINTER**. Printer.
- **B_CPU**. CPU device.
- **B_WORM**. Write-once, read-many device (such as CD-recordable).
- **B_CD**. CD-ROM.
- **B_SCANNER**. Scanner.
- **B_OPTICAL**. Optical device
- **B_JUKEBOX**. Jukebox device
- **B_NETWORK**. Network device

---

device_hooks

```
typedef struct {
    device_open_hook open;
    device_close_hook close;
    device_free_hook free;
    device_control_hook control;
    device_read_hook read;
    device_write_hook write;
    device_select_hook select;
    device_deselect_hook deselect;
    device_readv_hook readv;
    device_writev_hook writev;
    } device_hooks
```

This structure is used by device drivers to export their function hooks to the kernel.

---

device_icon

```
typedef struct {
    int32 icon_size;
    void *icon_data;
    } device_icon
```

When you want to obtain an icon for a specific device, call **ioctl()** on the open device, specifying the B_GET_ICON opcode. Pass in data a pointer to a device_icon structure in which **icon_size** indicates the size of icon you want and **icon_data** points to a buffer large enough to receive the icon's data.

icon_size can be either **B_MINI_ICON**, in which case the buffer pointed to by **icon_data** should be large enough to receive a 16x16 8-bit bitmap (256-byte), or **B_LARGE_ICON**, in which case the buffer should be large enough to receive a 32x32 8-bit bitmap (1024-byte). The most

obvious way to set up this buffer would be to create a BBitmap of the appropriate size and color depth and use its buffer, like this:

```
BBitmap bits(BRect(0, 0, B_MINI_ICON-1, B_MINI_ICON-1, 0, B_CMAP8));
device_icon iconrec;

iconrec.icon_size = B_MINI_ICON;
iconrec.icon_data = bits.Bits();
status_t err = ioctl(dev_fd, B_GET_ICON, &iconrec);

if (err == B_OK) {
   /* enjoy the icon */
   ...
   view->DrawBitmap(bits);
} else {
   /* I don't like icons anyway */
   }
```

driver_path

```
typedef char driver_path[256];
```

Used by the **B_GET_DRIVER_FOR_DEVICE** control function to return the pathname of the specified device.

open_device_iterator

```
typedef struct {
    uint32 cookie;
    char device[256];
    } open_device_iterator
```

Used by the **B_GET_NEXT_OPEN_DEVICE** control function. The first time you call this function, your **open_device_iterator** should have *cookie* initialized to 0. Then just keep calling it over and over; each time you'll get the name of the next open device. When an error is returned, you're done.

partition_info

```
typedef struct {
    off_t offset;
    off_t size;
    int32 logical_block_size;
    int32 session;
```

```
int32 partition;
char device[256];
} partition_info
```

The `partition_info` structure describes a disk partition, and is used by the `B_GET_PARTITION_INFO` and `B_SET_PARTITION` control commands.

The fields are:

- **`offset`** is the offset, in bytes, from the beginning of the disk to the beginning of the partition.
- **`size`** is the size, in bytes, of the partition.
- **`logical_block_size`** is the block size with which the file system was written to the partition.
- **`session`** and **`partition`** are the session and partition ID numbers for the partition.
- **`device`** is the pathname of the physical device on which the partition is located.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Writing Drivers

A device driver is an add-on that communicates with a specific device or type of device. Usually this communication involves some form of device-specific protocol. For example, an add-on that specifically addresses an Ethernet card or graphics card is a device driver. Likewise, add-ons that know how to talk to a class such as SCSI disks, ATA devices, ATAPI disks, or USB input devices is also a device driver.

A driver's job is to recognize the device and provide a means for applications to talk to it.

We can't stress this enough: a bug in a device driver can bring down the entire system. Be very careful, and be sure to test your work well.

To reduce the risk of the system being adversely affected by a bug in your code, you should put as much of your code into user space as possible.

This section covers the structure of device drivers, and provides some examples of how to write them.

Symbols Drivers Export

The kernel communicates with drivers by calling certain known entry points, which the driver must implement and export. These entry points are:

- **init_hardware()**
  Called when the system is booted, to let the driver detect and reset the hardware.
- **init_driver()**
  Called when the driver is loaded, so it can allocate needed system resources.
- **uninit_driver()**
  Called just before the driver is unloaded, so it can free allocated resources.
- **publish_devices()**
  Called to obtain a list of device names supported by the driver.
- **find_device()**
  Called to obtain a list of pointers to the hook functions for a specified device.
- **api_version**
  This exported value tells the kernel what version of the driver API it was written to, and should always be set to **B_CUR_DRIVER_API_VERSION** in your source code.

---

init_hardware()

```
status_t init_hardware(void)
```

This function is called when the system is booted, which lets the driver detect and reset the hardware it controls. The function should return **B_OK** if the initialization is successful; otherwise, an appropriate error code should be returned. If this function returns an error, the driver won't be used.

---

init_driver()

```
status_t init_driver(void)
```

Drivers are loaded and unloaded on an as-needed basis. When a driver is loaded by devfs, this function is called to let the driver allocate memory and other needed system resources. Return **B_OK** if initialization succeeds, otherwise return an appropriate error code. <<<what happens if this returns an error?>>>

---

uninit_driver()

```
void uninit_driver(void)
```

This function is called by devfs just before the driver is unloaded from memory. This lets the driver clean up after itself, freeing any resources it allocated.

publish_devices()

```
const char **publish_devices(void)
```

Devfs calls `publish_devices()` to learn the names, relative to /dev, of the devices the driver supports. The driver should return a **NULL**-terminated array of strings indicating all the installed devices the driver supports. For example, an ethernet device driver might return:

```
static char *devices[] = {
   "net/ether",
   NULL
};
```

In this case, devfs will then create the pseudo-file **/dev/net/ether**, through which all user applications can access the driver.

Since only one instance of the driver will be loaded, if support for multiple devices of the same type is desired, the driver must be capable of supporting them. If the driver senses (and supports) two ethernet cards, it might return:

```
static char *devices[] = {
   "net/ether1",
   "net/ether2",
   NULL
};
```

find_device()

```
device_hooks *find_device(const char *name)
```

When a device published by the driver is accessed, devfs communicates with it through a series of hook functions that handle the requests.The `find_device()` function is called to obtain a list of these hook functions, so that devfs can call them. The device_hooks structure returned lists out the hook functions.

The device_hooks structure, and what each hook does, is described in the next section.

api_version

```
int32 api_version;
```

This variable defines the API version to which the driver was written, and should be set to **`B_CUR_DRIVER_API_VERSION`** at compile time. The value of this variable will be changed with every revision to the driver API; the value with which your driver was compiled will tell devfs how it can communicate with the driver.

Device Hooks

The hook functions specified in the device_hooks function returned by the driver's `find_device()` function handle requests made by devfs (and through devfs, from user applications). These are described in this section.

The structure itself looks like this:

```
typedef struct {
   device_open_hook open;
   device_close_hook close;
   device_free_hook free;
   device_control_hook control;
   device_read_hook read;
   device_write_hook write;
   device_select_hook select;
   device_deselect_hook deselect;
   device_readv_hook readv;
   device_writev_hook writev;
} device_hooks;
```

In all cases, return `B_OK` if the operation is successfully completed, or an appropriate error code if not.

---

open_hook()

status_t **open_hook**(const char **name*, uint32 *flags*, void ***cookie*)

This hook function is called when a program opens one of the devices supported by the driver. The name of the device (as returned by `publish_devices()`) is passed in name, along with the flags passed to the Posix **`open()`** function. *cookie* points to space large enough for you to store a single pointer. You can use this to store state information specific to the **`open()`** instance. If you need to track information on a per-**`open()`** basis, allocate the memory you need and store a pointer to it in `*cookie`.

---

close_hook()

status_t **`close_hook`**(void ***cookie*)

This hook is called when an open instance of the driver is closed using the **`close()`** Posix function. Note that because of the multithreaded nature of the BeOS, it's possible there may still be transactions pending, and you may receive more calls on the device. For that reason, you shouldn't free instance-wide system resources here. Instead, you should do this in

free_hook(). However, if there are any blocked transactions pending, you should unblock them here.

---

free_hook()

```
status_t free_hook(void **cookie)
```

This hook is called once all pending transactions on an open (but closing) instance of your driver are completed. This is where your driver should release instance-wide system resources. `free_hook()` doesn't correspond to any Posix function.

---

read_hook()

```
status_t read_hook(void *cookie, off_t position, void *data, size_t *len)
```

This hook handles the Posix **`read()`** function for an open instance of your driver. Implement it to read *len* bytes of data starting at the specified byte *position* on the device, storing the read bytes at *data*. Exactly what this does is device-specific (disk devices would read from the specified offset on the disk, but a graphics driver might have some other interpretation of this request).

Before returning, you should set *len* to the actual number of bytes read into the buffer. Return B_OK if data was read (even if the number of returned bytes is less than requested), otherwise return an appropriate error.

---

readv_hook()

```
status_t readv_hook(void *cookie, off_t position, const struct iovec *vec,
    size_t count, size_t *len)
```

This hook handles the Posix **`readv()`** function for an open instance of your driver. This is a scatter/gather read function; given an array of iovec structures describing address/length pairs for a group of destination buffers, your implementation should fill each successive buffer with bytes, up to a total of *len* bytes. The *vec* array has *count* items in it.

As with read_hook(), set *len* to the actual number of bytes read, and return an appropriate result code.

---

write_hook()

```
status_t write_hook(void *cookie, off_t position, void *data, size_t len)
```

This hook handles the Posix **`write()`** function for an open instance of your driver. Implement it to write *len* bytes of data starting at the specified byte *position* on the device, from the buffer

pointed to by *data*. Exactly what this does is device-specific (disk devices would write to the specified offset on the disk, but a graphics driver might have some other interpretation of this request).

Return **B_OK** if data was read (even if the number of returned bytes is less than requested), otherwise return an appropriate error.

---

writev_hook()

```
status_t writev_hook(void *cookie, off_t position, const struct iovec *vec,
    size_t count, size_t *len)
```

This hook handles the Posix **writev()** function for an open instance of your driver. This is a scatter/gather write function; given an array of iovec structures describing address/length pairs for a group of source buffers, your implementation should write each successive buffer to disk, up to a total of *len* bytes. The *vec* array has *count* items in it.

Before returning, set *len* to the actual number of bytes written, and return an appropriate result code.

---

control_hook()

```
status_t control_hook(void *cookie, uint32 op, void *data, size_t len)
```

This hook handles the **ioctl()** function for an open instance of your driver. The control hook provides a means to perform operations that don't map directly to either **read()** or **write()**. It receives the *cookie* for the open instance, plus the command code *op* and the *data* and *len* arguments specified by **ioctl()**'s caller. These arguments have no inherent relationship; they're simply arguments to **ioctl()** that are forwarded to your hook function. Their definitions are defined by the driver. Common command codes can be found in **be/drivers/Drivers.h**.

The *len* argument is only valid when **ioctl()** is called from user space; the kernel always sets it to 0.

---

select_hook() , deselect_hook()

```
status_t select_hook(void *cookie, uint8 event, uint32 ref, selectsync *sync)
status_t deselect_hook(void *cookie, uint8 event, uint32 ref, selectsync *sync)
```

These hooks are reserved for future use. Set the corresponding entries in your device_hooks structure to **NULL**.

Driver Rules

Keep the following rules in mind for each instance of your driver:

- **`open()`** will be called first, and no other hooks will be called until **`open()`** returns.
- **`close()`** may be called while other requests are pending. As previously mentioned, if you have blocked transactions, you must unblock them when **`close()`** is called. Further calls to other driver hooks my continue to occur after **`close()`** is called; however, you should return an error to any such requests.
- **`free()`** isn't called until all pending transactions for the open instance are completed.
- Multiple threads may be accessing the driver's hooks simultaneously, so be sure to lock and unlock where appropriate.

The Be Book Drivers Drivers Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Driver Settings API

Declared in: be/drivers/driver_settings.h
Summary: *more...*

If your driver is loaded before the file system for the disk on which your settings file resides, your driver might not be able to load its settings using Posix calls. Also, a robust method for reading settings files—even if they might have become corrupted—can help the system be more stable; if your driver crashes trying to read its settings, the entire system is in jeopardy.

The driver settings API provides easy, safe access to boolean and string settings, and is available to all drivers and modules. If your driver has more complex settings, the `get_driver_settings()` function is available to retrieve all your settings in a hierarchical tree.

The boot loader reads the settings files from the boot volume and passes them to the kernel for distribution to the drivers upon request. The boot loader also lets the user add to these settings at boot time; a line of the form "filename:parameters" in the advanced safe mode menu will add "parameters" to the end of the specified settings file. This can be used to change debugging information and to test different options while developing your driver.

Using the Driver Settings API

Using the API is very simple. Just follow these basic steps:

- Call `load_driver_settings()` to load the settings data.
- Use `get_driver_settings()` or `get_driver_parameter()` and `get_driver_boolean_parameter()` to read the settings.
- Call `unload_driver_settings()` when you're done.

The Settings File

Driver settings files are kept in **~/config/settings/kernel/drivers**.

The settings file is formatted like this:

- Words beginning with "#" indicate that the rest of the line should be treated as a comment.
- Parameters can have values and subparameters. A parameter has the following form in the settings file:
- name [value]* [{
  [parameter]*
  }] ['n',',']
- Where [...] indicates an optional part, and [...]* indicates an optional repeated part.
- Names and values may not contain spaces unless the spaces are preceded by a backslash ('') or the words are enclosed in quotes.

Here's an example settings file:

```
device 0 {
  attribute1 value
  attribute2 value
}
device 1 {
  attribute1 value
}
```

For this settings file, `get_driver_settings()` will return a pointer to the following tree:

```
driver_settings = {
  parameter_count = 2
  parameters = {
    name = "device"
    value_count = 1
    values = { "0" }
```

```
      parameter_count = 2
      parameters = {
        name = "attributes1"
        value_count = 1
        values = "value"
        parameter_count = 0
        parameters = NULL
      },
      {
        name = "attribute2"
        value_count = 1
        values = "value"
        parameter_count = 0
        parameters = NULL
      }
    },
    {
      name = "device"
      value_count = 1
      values = { "1" }
      parameter_count = 1
      parameters = {
        name = "attribute1"
        value_count = 1
        values = "value"
        parameter_count = 0
        parameters = NULL
      }
    }
}
```

Loading the Settings

To load the driver's settings, you need to call `load_driver_settings()`. For example, if your driver's name is "xr_joystick", you might do this:

```
void *handle = load_driver_settings("xr_joystick");
```

The handle is then used when calling the other driver settings functions, to indicate which driver's settings you want to reference. This opaque reference protects you against any future changes in the kernel.

Reading the Settings

There are three functions you can use to read driver settings:

- `get_driver_boolean_parameter()` returns a boolean parameter's value.
- `get_driver_parameter()` returns a string parameter's value.

- `get_driver_settings()` returns all the settings at once, encapsulated in a hierarchical format.

Reading a Boolean Parameter

Let's look at a simple driver that has one boolean parameter, "debug", that enables a special debug mode. The value of this parameter is represented in the settings file by a line "debug value", where value is either "true" or "false". By default, if there's no setting for the debug parameter, false should be assumed. If the parameter is specified but no value is included, we want to assume that the user means true.

Our code to read this setting looks like this:

```
void *handle = load_driver_settings("xr_joystick");
bool debug = get_driver_boolean_parameter(handle, "debug", false,
            true);
unload_driver_settings(handle);
```

If there's no settings file, `load_driver_settings()` will return **`NULL`**. In this case, `get_driver_boolean_parameter()` will return **`false`** (the value we're passing as the *unknownValue* argument).

If there's a settings file, but the debug entry isn't found, the *unknownValue* argument is returned. Even though the handle is valid, the function can't find a value for that argument, so it uses this as the default.

If the file contains a line starting with "debug", the second word on the line is used as the value. If no value is specified, **`true`** is returned (the value of the *noArgValue* argument to `get_driver_boolean_parameter()`). Otherwise the following is done:

- If the value is "1", "true", "yes", "on", "enable", or "enabled", true is returned.
- If the value is "0", "false", "no", "off", "disable", or "disabled", false is returned.
- If the value matches none of these strings, it's treated as if no entry were found, and *unknownValue* is returned.

If more than one line containing the word "debug" is found, the last one in the file is used. This lets the user override, at boot time, the value previously specified in the settings file.

Reading a String Parameter

Reading string parameters works in much the same way, using the `get_driver_parameter()` function. The only difference is that the string returned will be **`NULL`** if the parameter is missing, or the file doesn't exist.

Reading All Parameters

If your driver has more complex parameters (such as parameters with multiple values, or with subparameters), you can read the entire settings tree using the **get_driver_settings()** function.

The **driver_settings** structure contains the root of the settings tree:

```
typedef struct driver_settings {
   int parameter_count;
   struct driver_parameter *parameters;
};
```

Each parameter is described by the driver_parameter structure:

```
typedef struct driver_parameter {
   char *name;
   int value_count;
   char **values;
   int parameter_count;
   struct driver_parameter *parameters;
};
```

C Functions

get_driver_boolean_parameter()

```
bool get_driver_boolean_parameter(void *handle,
    const char *keyName,
    bool unknownValue,
    bool noArgValue)
```

Returns the value of a given boolean parameter. The driver settings file is specified by the *handle*, as returned by `load_driver_settings()`. The parameter's name is given by *keyName*. If the parameter isn't found, *unknownValue* is returned. If the parameter exists but has no value, *noArgValue* is returned. This lets you easily deal with these two conditions, providing appropriate default values without additional code to check for error conditions.

If the handle is `NULL`, *unknownValue* is returned.

get_driver_parameter()

```
const char *get_driver_parameter(void *handle,
    const char *keyName,
    const char *unknownValue,
    const char *noArgValue)
```

Returns the value of a given string parameter. The driver settings file is specified by the *handle*, as returned by `load_driver_settings()`. The parameter's name is given by *keyName*. If the parameter isn't found, *unknownValue* is returned. If the parameter exists but has no value, *noArgValue* is returned. This lets you easily deal with these two conditions, providing appropriate default values without additional code to check for error conditions.

The special keyName value `B_SAFEMODE_SAFE_MODE` can be used if you want to find out whether or not BeOS was booted in safe mode; the value will be true if BeOS is running in safe mode, or false if a normal boot was performed.

If the handle is `NULL`, *unknownValue* is returned.

get_driver_settings()

```
const driver_settings *get_driver_settings(void *handle)
```

Returns the values of all parameters in encapsulated form.

load_driver_settings() , unload_driver_settings()

void ***load_driver_settings(**const char ****driverName***)**
status_t **unload_driver_settings(**void ****handle***)**

`load_driver_settings()` loads the settings for the driver specified by *driverName*, and returns a handle that should be used for calls to other driver settings functions. If you want to access the safe mode settings, pass **`B_SAFEMODE_DRIVER_SETTINGS`** Returns **`NULL`** if no settings are available for the driver.

`unload_driver_settings()` unloads the settings for the driver whose settings file is specified by *handle*. You should always call this function when you're done reading the settings.

Defined Types

driver_parameter

```
typedef struct driver_parameter {
        char *name;
        int value_count;
        char **values;
        int parameter_count;
        struct driver_parameter *parameters;
    };
```

Describes a subtree of parameters.

driver_settings

```
typedef struct driver_settings {
        int parameter_count;
        struct driver_parameter *parameters;
    };
```

Encapsulates all the settings for a driver.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Driver Settings API Function Summary

Function Summary

Function	Description
get_driver_boolean_parameter()	Returns a boolean parameter from a settings file
get_driver_parameter()	Returns a the value of the specified parameter
get_driver_settings()	Returns the values of all parameters in encapsulated form
load_driver_settings()	Loads the settings for the specified driver
unload_driver_settings()	Unloads the settings for the specified driver

---

The Be Book **Drivers** **Drivers Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Device Drivers

Writing device drivers requires additional knowledge of the inner workings of the BeOS. To write a driver you must follow the rules laid out in this chapter very carefully. These rules are not the same as those for writing a normal application—if your driver tries to do things it's not allowed to do, it could bring down the system.

This introduction covers how drivers interact with the kernel.

The Kernel and the Driver Author

The BeOS kernel comprises the basic functionality of the operating system: It knows how to start the boot process and to manage memory and threads, and it contains the PCI bus manager, the ISA bus manager, the device file system (`devfs`, which manages **/dev**), the root file system (`rootfs`, which manages **/**), and a few other things.

But this isn't enough to satisfy the needs of most applications, so the kernel uses add-ons to provide additional functionality. During the boot process, add-ons are loaded to handle "real" file systems, devices, busses, and the like.

Although Be's kernel add-ons provide support for a wide range of hardware—from disk devices to joysticks—this support isn't all-inclusive. Hardware developers may need to create their own drivers for their products.

Types of Kernel Add-on

There are three types of kernel add-on:

- *Device drivers* are add-ons that communicate directly with devices.
- *Modules* are kernel space add-ons that export an API for use by drivers (or by other modules).
- *File systems* are add-ons that support specific file systems, such as BFS, DOSFS, HFS, and so forth.

Device drivers and file systems, while extending the functionality of the kernel, are still accessible from user space: Applications can open and address them using file descriptors. Modules, on the other hand, are kernel-only units. Applications have no access to them; they're provided strictly for use by the kernel and other kernel add-ons.

Device Drivers

A device driver is an add-on that recognizes a specific device (or class of devices) and provides a means for the rest of the system to communicate with it. Usually this communication involves some form of device-specific protocol. For example, if the system wants to use an Ethernet card or graphics card, it needs to load a device driver add-on that knows how to communicate with that card. Similarly, code that knows how to talk to a class of devices (SCSI disks, ATA devices, ATAPI disks, or USB input devices, etc.) must be implemented as a device driver add-on.

Modules

Modules provide a uniform API for use by other modules and drivers. A module is like a library in that it acts as a repository for common code that's shared among several drivers.

For example: Let's say you have a device driver that talks to a SCSI device connected to a SCSI bus. A computer can have multiple SCSI busses. Because all SCSI devices use the same command set independent of the particular controller used to send the commands, the command set can be (and is) implemented as a module. The SCSI module knows how to handle all SCSI cards the BeOS supports; the API that the SCSI module defines is adopted by and augmented by the modules for specific SCSI device types (hard disks, scanners, CD drives, etc). The SCSI device modules are managed by a SCSI bus manager module, which knows how to cope with multiple busses and presents them in encapulated form to the drivers. The drivers then only need to deal with the bus manager's API, which makes the life of a driver author much simpler.

Be provides bus managers for SCSI, USB, IDE, and PCMCIA.

File Systems

File system add-ons provide support for disk and network file systems, such as BFS, HFS, FAT, ISO 9660, CIFS, and so forth. By creating new file system add-ons, developers can provide access to disks that are formatted using other file system.

Interactions with the Kernel

The kernel provides a number of services that drivers and modules can use. These include:

- Enabling and disabling interrupts.
- Setting up memory for DMA transactions.
- Access to other devices and modules.

The kernel also provides, at the user level, a Posix-like API for accessing devices. An application can open a device through **`open()`**, and use **`read()`**, **`write()`**, and **`ioctl()`** to access the device.

The Posix functions are converted into system calls into the kernel, which then passes them, via **`devfs`**, to the appropriate device driver.

---

devfs

The kernel manages device drivers through **devfs**, the device file system that's mounted at **/dev** during the boot process. In order to be accessed, a driver must "publish" itself by adding an entry in the **/dev** hierarchy. The basic Posix I/O functions (**open()**, **read()**, **write()**, **readv()**, **writev()**, **ioctl()**, and **close()**) can then be used.

Devfs makes the drivers available as needed in **/dev**; this usually happens the first time a program iterates through the directory entries for a subdirectory in **/dev**. The kernel knows where in the **/dev** hierarchy to publish drivers based on their location in **/boot/beos/system/add-ons/kernel/drivers/dev**. For example, the ATAPI driver publishes drivers in **/dev/disk/ide/atapi**, the driver is located in **/boot/beos/system/add-ons/kernel/drivers/dev/disk/ide/atapi**. Whew.

You can see this device hierarchy by using the "ls" command from a Terminal window. "ls /dev" will show you the root of the device hierarchy, "ls /dev/disk" will show you disk device busses, "ls /dev/disk/ide" will show you the IDE devices, and so forth.

In reality, drivers tend to publish themselves in multiple locations in the **/dev** hierarchy, so instead of putting duplicate copies of the driver in the **.../drivers/dev** tree, the driver binaries are put at **/boot/beos/system/add-ons/kernel/drivers/bin**, and symlinks are created in the **.../drivers/dev** tree at the appropriate place. (The same is also done for drivers in **/boot/home/config/add-ons/kernel/drivers/...**.)

Driver Implementation Principles

Much of the stability of the BeOS is achieved by constructing a nearly impenetrable wall between the kernel and user applications. Drivers are chinks in that wall. If a driver misbehaves or fails, there's a strong possibility that it will cause unexpected behavior or kill the entire system. It's absolutely critical that drivers not only be very carefully tested before being released to the public, but that they follow the rules to the letter.

Kernel Space vs. User Space

One way you can reduce the risk of your driver causing a general system failure is by putting as much code as possible in user space. Create a driver that loads into kernel space just enough code to handle the low-level interactions that absolutely have to be done in kernel space, then load code into user space to handle the rest of the work. If the add-on fails, the system will keep running—only your driver will fail.

Another plus to placing as much of your code as possible into user space is that it's much easier to debug code running in user space. Conventional debugging techniques that don't work for kernel code can be applied, and there's much less chance of taking down the system in the process.

Code Synchronization

Normally, spinlocks are a bad thing. A spinlock is a tight loop that watches for a condition to occur, looping endlessly until that condition is met (this is called "busy waiting"). This wastes valuable processor time, and is normally discouraged.

In general, you're encouraged to use semaphores instead of spinlocks; however, you can't acquire a semaphore while handling an interrupt. So if you need to synchronize code while handling an interrupt, you must use a spinlock. Put simply:

- Use spinlocks to protect critical sections in interrupt-handling code.
- Use semaphores in any other situation that calls for code synchronization.

Anywhere you use a spinlock to protect a critical section, you should disable interrupts. Of course, in an interrupt handler, you know that interrupts are already disabled, so you don't need to explicitly disable interrupts yourself. Interrupt handlers include I/O interrupts installed using **`install_io_interrupt()`** and timer interrupts installed by calling **`add_timer()`**.

Functions Available During Spinlocks

While your spinlock is running, you can perform the following actions. If it's not on this list, you can't do it.

- You can examine and alter hardware registers by using the appropriate bus manager hooks.
- You can examine and alter any locked-down memory.

- You can call the following kernel functions: `system_time()`, `atomic_add()`, `atomic_or()`, `atomic_and()`.
- You can call the following bus manager functions: `read_io_*()` and `write_io*()`.

If you do anything else inside your spinlock, you're breaking the rules, so don't do it.

Using Spinlocks

You need to be sure that your calls to `acquire_spinlock()` and `release_spinlock()` are balanced. In addition, if you nest spinlocks, they must be released in logical order—that is, in the opposite order in which they're acquired.

The kernel keeps track of which spinlocks are being held and which are being waited upon. The kernel assumes that spinlocks are initialized to 0, and then acquired and released in logical order.

By keeping track of spinlocks, the kernel can detect and break deadlocks on multiprocessor systems.

Disabling Interrupts

The only time you should ever disable interrupts in a device driver is just before entering a spinlock-protected critical section. There is absolutely no other reason to do it, so don't.

After disabling interrupts, you should reenable them as quickly as possible. You must **never**, under any circumstances, leave interrupts disabled for more than 50 microseconds. This means that your interrupt handler code (which runs with interrupts implicitly disabled) must execute in 50 microseconds or less.

Functions Available While Interrupts Are Disabled

If you have interrupts disabled and aren't in a spinlock, you can do the following things in addition to those listed above in "Functions Available During Spinlocks":

- You can call `release_sem_etc()` with the `B_DO_NOT_RESCHEDULE` flag set.
- You can call `get_sem_count()`, `add_timer()`, `cancel_timer()`, and `dprintf()`.

If you feel that you need to call a function not explicitly listed as permitted here, please contact Be Developer Support at devsupport@be.com and explain your needs; we'd be happy to discuss the situation with you.

Don't Block

It's crucial that your interrupt handler never block, whether directly (by acquiring a semaphore, for example) or indirectly (by calling a function that might block).

Blocking can happen in a surprisingly large number of BeOS functions. It's obvious that `acquire_sem()` can block, but you might not be aware that functions such as `malloc()` or

`read_port()` can block. Even touching unlocked memory areas can block because of virtual memory hits.

The point is this: If the BeOS function you want to call isn't explicitly listed in this section as one you can use, **don't call it**.

Don't Preempt

Your interrupt handler or spinlock section can't be preempted. Preemption could occur if you call `release_sem()` or `release_sem_etc()` without specifying the `B_DO_NOT_RESCHEDULE` flag. Normally, `release_sem()` lets the scheduler preempt your thread to allow other threads to acquire the semaphore as fast as possible. By specifying `B_DO_NOT_RESCHEDULE`, you tell the scheduler to allow your thread to continue running after it releases the semaphore.

If your interrupt handler wants to ensure that any preemption is handled immediately, it should specify `B_DO_NOT_RESCHEDULE` when calling `release_sem()`, then return `B_INVOKE_SCHEDULER`. This causes the scheduler to immediately handle preemption after your interrupt handler returns, instead of resuming the interrupted task. This is especially useful if your code called `release_sem_etc()` to release a semaphore that will allow other code to run elsewhere (such as in your driver's corresponding user-space code).

Again, when you call `release_sem_etc()`, be sure to specify the `B_DO_NOT_RESCHEDULE` flag to avoid any chance of preemption.

In summary, the order in which you should do things is this:

- Disable interrupts.
- Acquire the spinlock.
- Perform your tasks.
- Release the spinlock.
- Restore the original interrupt state.

File I/O

Sometimes a driver needs to be able to access disk files. Perhaps the driver has a preference file it needs to read. There are two ways to do this. You can use Posix I/O calls, or you can use the driver settings API provided by BeOS. The latter is preferred.

Using Posix Calls

Under BeOS, device drivers can access disk files using the standard low-level Posix I/O functions: `open()`, `close()`, `read()`, `write()`, and so forth. There aren't any special chores to attend to beforehand. Just `open()` the file and do your thing.

Two Posix extensions that might be helpful when you're writing code to perform file I/O from a device driver: `readv()` and `writev()`.

```
int readv(int fd, const struct iovec *vector, size_t count);

int writev(int fd, const struct iovec *vector, size_t count);

struct iovec {
   __ptr_t iov_base;
   size_t iov_len;
};
```

These functions provide a means to read and write contiguous portions of a file from multiple buffers. *vector* is a pointer to an array containing *count* vector records, each of which contains a pointer to a buffer, and the size of the buffer. **`readv()`** fills these buffers with data from the file, and **`writev()`** writes them to the file, in order.

When successful, **`readv()`** returns the number of bytes read.

For example, if your code needs to write two separate 1k buffers into a file, one after the other, you might do something like this:

```
struct iovec v[2];
v[0].iov_base = &buffer1;
v[0].iov_len = 1024;
v[1].iov_base = &buffer2;
v[1].iov_len = 1024;
if (writev(fd, &v, 2) != B_OK) {
   /* error */
}
```

Performing vectored I/O like this is often faster than doing multiple calls to **`read()`** and **`write()`**.

The Driver Settings API

If your driver is loaded before the file system for the disk on which your settings file resides, your driver might not be able to load its settings using Posix calls. The driver settings API lets you work around this circumstance. See "Driver Settings API" for details.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Time Information

Declared in: be/kernel/OS.h
Library: libroot.so
Summary: *more...*

The following functions set and get the system clock.

Time Functions

real_time_clock() , real_time_clock_usecs() , set_real_time_clock()

uint32 **real_time_clock** (void)
bigtime_t **real_time_clock_usecs** (void)
void **set_real_time_clock** (int32 *secs_since_jan1_1970*)

real_time_clock() returns the number of seconds that have elapsed since January 1, 1970.

real_time_clock_usecs() measures the same time span in microseconds.

set_real_time_clock() sets the value that the other two functions refer to.

system_time()

bigtime_t **system_time**(void)

Returns the number of microseconds that have elapsed since the computer was booted.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Time Information Function Summary

Function Summary

Function	Description
real_time_clock()	Returns seconds elapsed since January 1, 1970
real_time_clock_usecs()	Returns microseconds elapsed since January 1, 1970
set_real_time_clock()	Sets the system clock
system_time()	Returns the microseconds elapsed since the computer was booted

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Using Modules

Modules provide a means for multiple drivers to share common functionality; for example, if a variety of types of device might be accessed on the same bus, a module might be created to provide a common interface to the bus.

Your driver can access these modules via the kernel functions **`get_module()`** and **`put_module()`**, which obtain and release references to a specified module. When you call **`get_module()`**, you obtain a structure that provides information about the module, plus pointers to the module's functions. The module is defined in a header file provided by the module's author, similar to this:

```
#define MY_MODULE_NAME "generic/mymodule/v1"

struct my_module_info {
   module_info module;
   int32 (*function1)();
   int32 (*function2)();
   void (*configure)(int32 parameter, int32 value);
};
```

When you want to access the module's functions, you call **`get_module()`** to get a pointer to this structure from the kernel:

```
struct my_module_info *minfo = NULL;

/* get a pointer to the module */

get_module(MY_MODULE_NAME, (module_info **) &minfo);
```

Once you've done this, you can call the module's functions through the structure:

```
minfo->configure(0, 10);
```

When you're done with the module, you should call **`put_module()`** to release it. The kernel loads and unloads modules as needed, and properly calling **`put_module()`** lets the kernel do its job.

```
put_module(MY_MODULE_NAME);
```

If you want a better understanding of how modules work, see the "Writing Modules" section.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Writing Modules

Modules provide services that can be used by other modules, by device drivers, and by the kernel itself. They can be dynamically loaded and unloaded by the kernel, as needed. If a client can't find a module it needs, it will still load, which gives it the opportunity to find another way to perform the desired tasks, or to disable those features of itself.

Modules, like drivers, export an API through a structure that provides pointers to the functions provided by the module, along with other information about the module. You do this by expanding upon the basic module definition in **be/drivers/module.h**. For example, you might define your module information structure like this:

```
#define MY_MODULE_NAME "generic/mymodule/v1"

struct my_module_info {
   module_info module;
   int32 (*function1)();
   int32 (*function2)();
   void (*configure)(int32 parameter, int32 value);
};
```

Note that the first field in your module information structure is a module_info, which looks like this:

```
struct module_info {
   const char *name;
   uint32 flags;
   status_t (*std_ops);
};
```

The *name* field should be a pointer to the driver's name as indicated in your module's header file (in this example, `MY_MODULE_NAME`).

The *flags* field specifies which flags should be in effect for your module. Currently, the **`B_KEEP_LOADED`** flag is the only one available; as expected, it tells the kernel not to unload your module when nobody is using it; normally, the first time your module is requested by someone calling **`get_module()`**, the kernel loads it. With each subsequent call to **`get_module()`**, a reference count is incremented. Every time **`put_module()`** is called to release the module, the reference count is decremented. When the counter reaches zero, the module is unloaded. **`B_KEEP_LOADED`** prevents unloading from taking place.

std_ops is a pointer to a function that your module must provide. This function is called to handle standard module operations. Currently, there are only two of these operations (initialization and uninitialization). Your module's **`std_ops()`** function will probably look something like this:

```
```

```
static status_t std_ops(int32 op, ...) {
   switch(op) {
      case B_MODULE_INIT:
         /* do whatever you need to do */
         break;
      case B_MODULE_UNINIT:
         /* do whatever you need to do */
         break;
      default:
         return B_ERROR;   /* necessary, for future expansion */
   }
   return B_OK;
}
```

It's important to return **B_ERROR** for any unknown operations, in case future versions of the kernel define additional operations.

Exporting your module to the outside world is similar to publishing device driver hooks, but since you define the hooks yourself, it's slightly more involved. Your module needs to have a filled-out version of your module's information structure, like this:

```
static struct my_module_info my_module {
   {
      MY_MODULE_NAME,   /* module name */
      0,   /* flags */
      std_ops
   },
   function1,
   function2,
   configure
};
```

When loading your module, the kernel looks for a symbol called "modules" that contains a list of pointers to the modules you export, terminated by a **NULL**:

```
_EXPORT module_info *modules[] = {
   (module_info *) &my_module,
   NULL
};
```

This is how the kernel finds out what modules are available for use by drivers (or by other modules). See the "Using Modules" section for details on how modules are accessed by other drivers or modules.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

- The area_malloc Module

Function Summary

Function Summary

Function	Description
calloc()	Allocates memory from a pool, erasing the bytes to zero
create_pool()	Creates a new memory pool
delete_pool()	Deletes a memory pool
free()	Releases a block of memory from a pool
malloc()	Allocates memory from a pool
realloc()	Changes the size of the specified memory block, possibly moving it

---

The Be Book **Drivers** **Drivers Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaRoster

Derived from: BLooper
Declared in: be/media/MediaRoster.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BMediaRoster class comprises the functionality that applications that use the Media Kit can access.

An application can only have a single instance of the BMediaRoster class, which is accessed by calling the static member function **BMediaRoster::Roster()**, which creates the media roster, establishes the connection to the Media Server, then returns a pointer to the roster.

The creation of the roster object is thread protected, so you can safely call **BMediaRoster::Roster()** from multiple threads without synchronization, and both threads will safely get the same instance. The cost of this synchronization is low enough that there's no need to cache the returned pointer, but it's perfectly safe to do so if you wish:

```
BMediaRoster *gMediaRoster;

int main(void) {
   status_t err;

   BApplication app("application/x-vnd.me-myself");
   gMediaRoster = BMediaRoster::Roster(&err);

   if (!gMediaRoster || (err != B_OK)) {
      /* the Media Server appears to be dead -- handle that here */
   }

   /* The Media Server connection is in place -- enjoy! */

   return 0;
}
```

Because BMediaRoster is derived from BLooper, you should create your BApplication before calling **BMediaRoster::Roster()**, although the BApplication doesn't have to be running yet.

You should never delete the BMediaRoster returned to you by **BMediaRoster::Roster()**. Also, you can't derive a class from BMediaRoster.

If you want to receive notifications from the Media Server when specific changes occur, such as nodes coming online or going offline, for example, you can register to receive such

notifications by calling **StartWatching()**.

Playing Media from Disk

To play a media file from disk, you would follow the following steps:

- Create an entry_ref that refers to the file to be played.
- Call `SniffRef()` to obtain a dormant_node_info reference to the node that is best capable of playing back the media file.
- Call `InstantiateDormantNode()` to instantiate the node; this returns a media_node that you can use for other BMediaRoster calls.
- Use `SetRefFor()` to pass the entry_ref of the file to be played to the node.
- Call **`GetFreeOutputs()`** to obtain a media_output structure describing an available output on the producer node. This structure contains a media_source you can pass to `Connect()` as the source of the media data.
- The media_output structure returned by `GetFreeOutputsFor()` contains a media_format field that indicates the general type of media data contained by the media file; if you don't already know what type of data you're playing back (audio or video or whatever), this will let you determine what you'll be playing.
- You can use this value to determine what type of destination is needed to output the data (for example, **`B_MEDIA_RAW_AUDIO`** would go to an audio output, and **`B_MEDIA_RAW_VIDEO`** would go to a video output).
- Call `GetAudioMixer()` to get a media_node for the default audio mixer (if the media data is audio), or `GetVideoOutput()` to get a media_node for the default video output (if the media data is video).
- Use `GetFreeInputsFor()` to obtain a media_input structure describing an available input on the consumer node (the audio mixer or video output); this structure contains a media_destination you can pass to `Connect()`.
- Call `Connect()` to connect the source to the destination.
- Next, you need to set the time source for the media file's node. Normally you'll set this to the preferred time source. The audio mixer or video output is already slaved to an appropriate time source.
- Finally, you can use `StartNode()` to start the producer, consumer, and time source. The media file should begin playing back.

Let's look at actual sample code that does this. This example is particular to playing movie files; in particular, it assumes that the video is encoded (**`B_MEDIA_ENCODED_VIDEO`**) and that the audio is in a raw audio format (**`B_MEDIA_RAW_AUDIO`**). However, it demonstrates the principles of playing both encoded and raw media formats, and you can easily extrapolate from it any type of playback you need. First it's necessary to identify an appropriate node to handle the file, and to instantiate the node and configure it for the file we want to play:

```
bigtime_t      duration;
media_node      timeSourceNode;

media_node      mediaFileNode;
media_output   fileNodeOutput;
int32         fileOutputCount;

media_output   fileAudioOutput;
int32         fileAudioCount;

media_node      codecNode;
media_output   codecOutput;
media_input      codecInput;

media_node      videoNode;
media_input      videoInput;
int32         videoInputCount;

media_node      audioNode;
media_input      audioInput;
int32         audioInputCount;

dormant_node_info nodeInfo;
status_t err;

playingFlag = false;
roster = BMediaRoster::Roster();

initStatus = roster->SniffRef(*ref, 0, &nodeInfo);
if (initStatus) {
   return;
}

initStatus = roster->InstantiateDormantNode(nodeInfo, &mediaFileNode);
if (initStatus) {
   return;
}

roster->SetRefFor(mediaFileNode, *ref, false, &duration);
if ((err = Setup()) != B_OK) {
   printf("Error %08lX in Setup()n", err);
}
else {
   Start();
}
```

This code begins by obtaining a pointer to the media roster. It then calls `SniffRef()` to get a `dormant_node_info` structure describing the best-suited node for reading the media data from the file specified by `ref`.

Once a `dormant_node_info` structure has been filled out, the `InstantiateDormantNode()` function is called to instantiate a node to handle the file. A dormant node is a node whose code resides in an add-on, instead of within the application itself. The `nodeInfo` structure is passed into the function, and on return, the `mediaFileNode` has been set up for the appropriate node.

Since the media_node `mediaFileNode` is a file handling node, the `SetRefFor()` function is then called to tell the newly-instantiated file handler node what file it should handle. The inputs here are:

- `mediaFileNode` is the node whose file reference is to be set.
- `ref` is the file that the node should reference.
- **`false`** indicates that the file must already exist. If this flag were true, and the file indicated by ref were nonexistent, the node would create a new file. Since we're playing a file, we don't want to do that.
- `duration` is a bigtime_t variable that will receive the duration of the media file, in microseconds.

Once this has been accomplished, it's time to instantiate the other nodes needed to perform the media playback. Note that your code should check the error results from each of these calls and only proceed if `B_OK` is returned.

The `Setup()` and `Start()` functions used in the example above are given below. `Setup()` actually sets up the connections and instantiates the various other nodes (such as codecs and output nodes) required to play back the media data. Let's take a look at `Setup()` next:

```
status_t MediaPlayer::Setup(void) {
   status_t err;
   media_format tryFormat;
   dormant_node_info nodeInfo;
   int32 nodeCount;

   err = roster->GetAudioMixer(&audioNode);
   err = roster->GetVideoOutput(&videoNode);
```

First, `GetAudioMixer()` and `GetVideoOutput()` are called to obtain an audio mixer node and a video output node. The nodes returned by this function are based on the user's preferences in the Audio and Video preference applications. By default, video is output to a simple video output consumer that creates a window to contain the video display.

The VideoConsumer node will be available with R4.5; it's not provided in R4. In addition, there are no video producer nodes in R4; media add-ons for a variety of movie file formats will also be available beginning with R4.5.

```
err = roster->GetTimeSource(&timeSourceNode);
b_timesource = roster->MakeTimeSourceFor(timeSourceNode);
```

This code obtains a media_node for the preferred time source, and then creates a BTimeSource object that refers to the same node; we'll need to be able to make some BTimeSource calls to obtain some specific timing information later.

A time source is a node that can be used to synchronize other nodes. By default, nodes are slaved to the system time source, which is the computer's internal clock. However, this time source, while very precise, isn't good for synchronizing media data, since its concept of time has nothing to do with actual media being performed. For this reason, you typically will want to change nodes' time sources to the preferred time source.

You can think of a media node (represented by the **media_node** structure) as a component in a home theater system you might have at home. It has inputs for audio and video (possibly multiple inputs for each), and outputs to pass that audio and video along to other components in the system. To use the component, you have to connect wires from the outputs of some other components into the component's inputs, and the outputs into the inputs of other components.

The Media Kit works the same way. We need to locate audio outputs from the `mediaFileNode` and find corresponding audio inputs on the `audioNode`. This is analogous to choosing an audio output from your new DVD player and matching it to an audio input jack on your stereo receiver. Since you can't use ports that are already in use, we call **GetFreeOutputsFor()** to find free output ports on the `mediaFileNode`, and **GetFreeInputsFor()** to locate free input ports on the `audioNode`.

```
err = roster->GetFreeOutputsFor(mediaFileNode, &fileAudioOutput, 1,
            &fileAudioCount, B_MEDIA_RAW_AUDIO);
err = roster->GetFreeInputsFor(audioNode, &audioInput, fileAudioCount,
            &audioInputCount, B_MEDIA_RAW_AUDIO);
```

We only want a single audio connection between the two nodes (a single connection can carry stereo sound), and the connection is of type **B_MEDIA_RAW_AUDIO**. On return, `fileAudioOutput` and `audioInput` describe the output from the `mediaFlieNode` and the input into the `audioNode` that will eventually be connected to play the movie's sound.

We likewise have to find a video output from the `mediaFileNode` and an input into the `videoNode`. In this case, though, we expect the video output from the `mediaFileNode` to be encoded, and the `videoNode` will want to receive raw, uncompressed video. We'll work that out in a minute; for now, let's just find the two ports:

```
err = roster->GetFreeOutputsFor(mediaFileNode, &fileNodeOutput, 1,
            &fileOutputCount, B_MEDIA_ENCODED_VIDEO)
err = roster->GetFreeInputsFor(videoNode, &videoInput, fileOutputCount,
            &videoInputCount, B_MEDIA_RAW_VIDEO);
```

The problem we have now is that the `mediaFileNode` is outputting video that's encoded somehow (like in Cinepak format, for instance). The `videoNode`, on the other hand, wants to display raw video. Another node must be placed between these to decode the video (much like having an adapter to convert PAL video into NTSC, for example). This node will be the codec that handles decompressing the video into raw form.

We need to locate a codec node that can handle the video format being output by the `mediaFileNode`. This is accomplished like this:

```
nodeCount = 1;
err = roster->GetDormantNodes(&nodeInfo, &nodeCount,
            &fileNodeOutput.format);
if (!nodeCount) {
   return -1;
}
```

This call to **`GetDormantNodes()`** looks for a dormant node that can handle the media format specified by the `mediaFileNode`'s output **`media_format`** structure. Information about the node is returned in `nodeInfo`. `nodeCount` indicates the number of matching nodes that were found. If it's zero, an error is returned.

Note that in real life you should ask for several nodes, and search through them, looking at the formats until you find one that best meets your needs.

Then we use **`InstantiateDormantNode()`** to instantiate the codec node, and locate inputs into the node (that accept encoded video) and outputs from the node (that output raw video):

```
err = roster->InstantiateDormantNode(nodeInfo, &codecNode);
err = roster->GetFreeInputsFor(codecNode, &codecInput, 1, &nodeCount,
            B_MEDIA_ENCODED_VIDEO);
err = roster->GetFreeOutputsFor(codecNode, &codecOutput, 1,
            &nodeCount, B_MEDIA_RAW_VIDEO);
```

Now we're ready to start connecting these nodes together. If we were setting up a home theater system, right about now we'd be getting rug burns on our knees and skinned knuckles on our hands, trying to reach behind the entertainment center to run wires. The Media Kit is way easier than that, and doesn't involve salespeople telling you to get expensive gold-plated cables.

We begin by connecting the file node's video output to the codec's input:

```
tryFormat = fileNodeOutput.format;
err = roster->Connect(fileNodeOutput.source, codecInput.destination,
            &tryFormat, &fileNodeOutput, &codecInput);
```

`tryFormat` indicates the format of the encoded video that will be output by the `mediaFileNode`. **`Connect()`**, in essense, runs a wire between the output from the media node's video output (`fileNodeOutput`) to the codec node's input.

You may wonder what's up with the `fileNodeOutput.source` and `codecInput.destination` structures. These **media_source** and **media_destination** structures are simplified descriptors of the two ends of the connection. They contain only the data absolutely needed for the Media Kit to establish the connection. This saves some time when issuing the **Connect()** call (and time is money, especially in the media business).

Next it's necessary to connect the codec to the video output node. This begins by setting up `tryFormat` to describe raw video of the same width and height as the encoded video being fed into the codec, then calling **Connect()** to establish the connection:

```
   tryFormat.type = B_MEDIA_RAW_VIDEO;
   tryFormat.u.raw_video = media_raw_video_format::wildcard;
   tryFormat.u.raw_video.display.line_width =
            codecInput.format.u.encoded_video.output.display.line_width;
   tryFormat.u.raw_video.display.line_count =
            codecInput.format.u.encoded_video.output.display.line_count;
   err = roster->Connect(codecOutput.source, videoInput.destination,
               &tryFormat, &codecOutput, &videoInput);
```

Now we connect the audio from the media file to the audio mixer node. We just copy the **media_format** from the file's audio output, since both ends of the connection should exactly match.

```
   tryFormat = fileAudioOutput.format;
   err = roster->Connect(fileAudioOutput.source, audioInput.destination,
               &tryFormat, &fileAudioOutput, &audioInput);
```

The last step of configuring the connections is to ensure that all the nodes are slaved to the preferred time source. This will keep them synchronized with the preferred time source (and by association, with each other):

```
   err = roster->SetTimeSourceFor(mediaFileNode.node, timeSourceNode.node);
   err = roster->SetTimeSourceFor(videoNode.node, timeSourceNode.node);
   err = roster->SetTimeSourceFor(codecOutput.node.node,
            timeSourceNode.node);
   err = roster->SetTimeSourceFor(audioNode.node, timeSourceNode.node);
   return B_OK;
}
```

Finally, we return **B_OK** to the caller. Note that this code should be enhanced to check the results of each BMediaRoster call, and to return the result code if it's not **B_OK**. This has been left out of this example for brevity.

The `Start()` function actually starts the movie playback. Starting playback involves starting, one at a time, all the nodes involved in playing back the audio. This includes the audio mixer (`audioNode`), the media file's node (`mediaFileNode`), the codec, and the video node.

```
status_t MediaPlayer::Start(void) {
   status_t err;
   err = roster->GetStartLatencyFor(timeSourceNode, &startTime);
   startTime += b_timesource->PerformanceTimeFor(BTimeSource::RealTime()
               + 1000000 / 50);

   err = roster->StartNode(mediaFileNode, startTime);
   err = roster->StartNode(codecNode, startTime);
   err = roster->StartNode(videoNode, startTime);

   return B_OK;
}
```

Because there's lag time between starting each of these nodes, we pick a time a few moments in the future for playback to begin, and schedule each node to start playing at that time. So we begin by computing that time in the future.

The **`BTimeSource::RealTime()`** static member function is called to obtain the current real system time. We add a fiftieth of a second to that time, and convert it into performance time units. This is the time at which the performance of the movie will begin (basically a fiftieth of a second from "now"). This value is saved in `startTime`. These are added to the value returned by **`GetStartLatencyFor()`**, which returns the time required to actually start the time source and all the nodes slaved to it.

Then we simply call **`BMediaRoster::StartNode()`** for each node, specifying `startTime` as the performance time at which playback should begin.

Again, error handling should be added to actually return the error code from these functions.

Stopping playback of the movie is even simpler:

```
   err = roster->StopNode(mediaFileNode, 0, true);
   err = roster->StopNode(codecNode, 0, true);
   err = roster->StopNode(videoNode, 0, true);
```

This tells the media file, video codec, and video output nodes to stop immediately. If we wanted them to stop together at some time in the future, we could compute an appropriate performance time and pass that instead of 0. In this case, we would need to specify false for the last argument; when this value is true, **`StopNode()`** stops the node immediately. We could use this ability to schedule all three nodes to stop at the same time, so that video and audio playback would halt simultaneously.

Note that we don't stop the audio mixer node. You should never stop the mixer node, because other applications are probably using it.

Once you're done playing the movie, and have stopped playback, you should disconnect the nodes from each other:

```
```

```
   err = roster->Disconnect(mediaFileNode.node, fileNodeOutput.source,
               codecNode.node, codecInput.destination);
   err = roster->Disconnect(codecNode.node, codecOutput.source,
               videoNode.node, videoInput.destination);
   err = roster->Disconnect(mediaFileNode.node, fileAudioOutput.source,
               audioNode.node, audioInput.destination);
```

This will close out the connections between the media file node and the video codec, the codec and the video output, and between the file node and the audio mixer. You should always stop playback before disconnecting; although nodes aren't allowed to crash if you disconnect them while running, their behavior isn't specified, and may not be what you expect.

Once the connections are severed, you should release any dormant nodes you instantiated. This includes not only nodes instantiated using **`InstantiateNodeFor()`**, but also default nodes (those obtained using functions like **`GetAudioInput()`** and **`GetVideoOutput()`**, for example):

```
   roster->ReleaseNode(codecNode);
   roster->ReleaseNode(mediaFileNode);
   roster->ReleaseNode(videoNode);
   roster->ReleaseNode(audioNode);
```

If you want to play audio, you may find it much easier to use the BSound and BSoundPlayer classes to do so. As of R4.5, there are no Be-provided nodes for producing audio from a disk file.

Detecting When Playback Is Complete

There isn't a Media Kit function that can directly tell you whether or not the media has reached the end of the data during playback. However, the following easy-to-implement code can do the job for you:

```
bigtime_t currentTime;
bool isPlaying = true;

currentTime = b_timesource->PerformanceTimeFor(BTimeSource::RealTime());
if (currentTime >= startTime+duration) {
   isPlaying = false;
}
```

This works by obtaining the time source's performance time and comparing it to the time at which playback of the movie was begun plus the movie's duration (both of which were saved when we initially set up and began playback of the movie, as seen in the code in the previous section above).

If the current performance time is equal to or greater than the sum of the starting time and the movie's duration, then playback is finished, and we set `isPlaying` to **`false`**; otherwise, this value remains **`true`**.

Using BMediaRoster Functions from Nodes

You can issue BMediaRoster function calls from within your own node, however, as a general rule, you shouldn't call BMediaRoster functions from within your control thread, or while the control thread is blocked. Many BMediaRoster functions use synchronous turnarounds, and will deadlock in this situation. You should assume, for safety's sake, that all BMediaRoster functions will deadlock if used in these cases.

For example, if you have an application that's playing video into a window, and you call StopNode() from the window's MessageReceived() function, a deadlock would occur if the video player node blocks waiting on the window to be unlocked, and the StopNode() function is keeping the window locked while it waits for the video producer node, which is blocked waiting on the consumer node, and so forth. Deadlock results, and that's a bad thing.

Instead, you should consider creating a seperate BLooper that manages your nodes. Future versions of the Media Kit will provide convenience classes to do some of this for you.

Constructor and Destructor

BMediaRoster()

```
BMediaRoster()
```

You never construct a BMediaRoster yourself. Instead, use the static `Roster()` function to obtain an instance of the BMediaRoster class that you can use.

~BMediaRoster

```
~BMediaRoster()
```

You never delete a BMediaRoster yourself. Just let it go away automatically when your application shuts down.

Member Functions

AudioBufferSizeFor()

```
ssize_t AudioBufferSizeFor(int32 channelCount,
    uint32 sampleFormat,
    float frameRate,
    bus_type busKind)
```

`AudioBufferSizeFor()` returns the size, in bytes, that the Media Kit recommends for audio data with *channelCount* channels, with the specified *sampleFormat* and *frameRate*.

The busKind argument is a bus_type value (see **drivers/config_manager.h**) indicating the type of bus the data is moving across. Specify **`B_UNKNOWN_BUS`** if you don't know.

Connect() , Disconnect()

```
status_t Connect(const media_source &source,
    const media_destination &destination,
    media_format *ioFormat,
    media_output *outOutput,
    media_input *outInput)
status_t Connect(const media_source &source,
    const media_destination &destination,
    media_format *ioFormat,
    media_output *outOutput,
    media_input *outInput,
    uint32 inFlags,
    void *_reserved = NULL)
```

```
status_t Disconnect(media_node_id sourceNode,
    const media_source &source,
    media_node_id destinationNode,
    const media_destination &destination)
```

`Connect()` negotiates a connection from the *source* to the *destination*, using the media format specified in *ioFormat* as a basis for the negotiation; *ioFormat* is changed to the negotiated format before this call returns. This describes the format of media data that will flow across the connection.

The actual connection is returned as an output and an input in *outOutput* and *outInput*. These two structures contain the data format as interpreted by the source and destination. There may be differences among these formats if wildcard fields were used in the original format.

The second form of `Connect()` lets you specify connect flags. Currently the only possible flag is **`B_CONNECT_MUTED`**, which indicates that the connection should be muted on creation.

The actual media_source and media_destination used for the connection may vary from those passed into `Connect()` if the *source* or the *destination* creates new sources or destinations for each connection request; the *outOutput* and *outInput* structures contain the actual media_source and media_destination values resulting from the call.

For more detailed information on the use of wildcards in format negotiation, see **`media_audio_format::wildcard`** and **`media_video_format::wildcard`**. A media_format with a type of **`B_MEDIA_UNKNOWN_TYPE`** matches any media class and format, although without specific knowledge of the source and destination, this will rarely result in a useful connection.

`Disconnect()` breaks the connection established between *source* and *destination*, which must belong to the nodes *sourceNode* and *destinationNode*, repectively.

The result of breaking a connection that's currently running is undefined, but is not permitted to crash. Your application should stop both nodes involved in a connection prior to disconnecting them.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error terminating the connection.
- **`B_NAME_NOT_FOUND`**. The connection couldn't be made.
- Other errors. The nodes that are being connected may return other error codes as they see fit.

Current see Roster()

GetAllInputsFor() , GetAllOutputsFor()

```
status_t GetAllInputsFor(const media_node &node,
    media_input *outInputs,
    int32 bufNumInputs
    int32 *outTotalCount)
```

```
status_t GetAllOutputsFor(const media_node &node,
    media_output *outOutputs,
    int32 bufNumOutputs
    int32 *outTotalCount)
```

`GetAllInputsFor()` fills the array of media_input structures specified by *outInputs* with information about all inputs belonging to *node*; the number of elements that *outInputs* can

hold is passed in *bufNumInputs*.

Similarly, `GetAllOutputsFor()` fills the array *outOutputs* with information about all outputs from the specified *node*.

Both functions return the number of elements actually returned in the buffer in *outTotalCount*. If this number is less than the number you requested, your buffer was too small to receive all the results of the query. In this case, you might want to resize your array and try again.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_NODE**. The node isn't of the correct type for the call you issued.
- Other errors. An error occurred communicating with the producer or with the Media Server.

GetAllOutputsFor() see GetAllInputsFor()

GetAudioInput() , GetVideoInput()

status_t **GetAudioInput(**media_node ***outNode*)**
status_t **GetVideoInput(**media_node ***outNode*)**

These functions return the nodes designated by the user as the preferred nodes for audio and video input. You can then query the returned node, hook into it, and manipulate it, using the reference returned in *outNode*.

Once your application has finished using these nodes (and they've been stopped and disconnected), you should release them by calling **ReleaseNode()**.

RETURN CODES

- B_OK. No error locating the default input node.
- **B_NAME_NOT_FOUND**. The default node couldn't be identified.

GetAudioMixer() see GetAudioOutput()

GetAudioOutput() , GetVideoOutput() , GetAudioMixer()

status_t **GetAudioOutput(**media_node ***outNode*)**

```
status_t GetAudioOutput(media_node *outNode, int32 *outInputID
    BString *outInputName)
status_t GetVideoOutput(media_node *outNode)
status_t GetAudioMixer(media_node *outNode)
```

These functions return the nodes designated by the user as the preferred nodes for audio and video output. You can then query the returned node, hook into it, and manipulate it, using the reference returned in *outNode*.

The second form of GetAudioOutput() returns additional information, including the input ID of the input used for audio output, and the input's name.

You should usually use `GetAudioMixer()` when getting a node for playing audio instead of using the `GetAudioOutput()` function. `GetAudioOutput()` returns the lower-level node for audio output, which you would typically only need access to if you wanted to do some form of processing on all audio data being played in the system (such as a level meter).

The `GetAudioMixer()` function returns a reference to the audio mixer, which will perform audio mixing, format conversion, and sample rate conversion for you, then pass along the audio to the output node.

Once your application has finished using these nodes (and they've been stopped and disconnected), you should release them by calling `ReleaseNode()`.

RETURN CODES

- B_OK. No error locating the default input node.
- `B_NAME_NOT_FOUND`. The default node couldn't be identified.

GetConnectedInputsFor() , GetFreeInputsFor()

```
status_t GetConnectedInputsFor(const media_node &node,
    media_input *outActiveInputsList,
    int32 numListInputs
    int32 *outNumInputs)
status_t GetFreeInputsFor(const media_node &node,
    media_input *outFreeInputsList,
    int32 numListInputs
    int32 *outNumInputs,
    media_type filterType = B_MEDIA_UNKNOWN_TYPE)
```

`GetConnectedInputsFor()` fills the array of media_input structures specified by *outActiveInputsList* with information about all inputs belonging to *node* that are currently connected to some output; the number of elements that *outActiveInputsList* can hold is passed in *numListInputs*.

Similarly, `GetFreeInputsFor()` fills the array *outFreeInputsList* with information about all inputs that are still available in the specified *node*. Specifying a *filterType* other than **`B_MEDIA_NO_TYPE`** lets you obtain a list of inputs for a specific media type (or for inputs that can handle any media type). This is especially useful if you're only interested in a list of accepted media types your application supports.

Even though a node may report that a specific number of free inputs are available, it is possible that a node might create more inputs on demand. There is no way to know if this might happen, so `GetFreeInputsFor()` may not tell you whether or not a node can accept all the connections you'd like to make.

Both functions return the number of elements actually returned in the buffer in *outNumInputs*. If this number is less than *numListInputs*, your buffer was too small to receive all the results of the query. In this case, you might want to resize your array and try again.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error cloning the node.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be cloned.

GetConnectedOutputsFor() , GetFreeOutputsFor()

```
status_t GetConnectedOutputsFor(const media_node &node,
    media_output *outActiveOutputsList,
    int32 numListOutputs
    int32 *outNumOutputs)
```

```
status_t GetFreeOutputsFor(const media_node &node,
    media_output *outFreeOutputsList,
    int32 numListOutputs
    int32 *outNumOutputs,
    media_type filterType = B_MEDIA_UNKNOWN_TYPE)
```

`GetConnectedOutputsFor()` fills the array of media_output structures specified by *outActiveOutputsList* with information about all outputs belonging to *node* that are currently connected to some input; the number of elements that *outActiveOutputsList* can hold is passed in *numListOutputs*.

Similarly, `GetFreeOutputsFor()` fills the array *outFreeOutputsList* with information about all outputs that are still available in the specified *node*. Specifying a *filterType* other than **`B_MEDIA_UNKNOWN_TYPE`** lets you obtain a list of outputs for a specific media type (or for outputs that can handle any media type). This is especially useful if you're only interested in a list of accepted media types your application supports.

Even though a node may report that a specific number of free outputs are available, it is possible that a node might create more outputs on demand. There is no way to know if this might happen, so `GetFreeOutputsFor()` may not tell you whether or not a node can accept all the connections you'd like to make.

Both functions return the number of elements actually returned in the buffer in *outNumOutputs*. If this number is less than *numListOutputs*, your buffer was too small to receive all the results of the query. In this case, you might want to resize your array and try again.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_NODE**. The node isn't a buffer producer.
- Other errors. An error occurred communicating with the producer or with the Media Server.

GetDormantFlavorInfoFor()

```
status_t GetDormantFlavorInfoFor(const dormant_node_info &inDormantNode,
    dormant_flavor_info *outFlavor)
```

This function returns, in *outFlavor*, information describing the dormant flavors supported by the dormant node *inDormantNode*.

RETURN CODES

- B_OK. No error.
- Messaging errors. An error occurred communicating with the Media Server.

GetDormantNodes()

```
status_t GetDormantNodes(dormant_node_info *outDormantNodeList,
    int32 *inOutNumNodes,
    const media_format *hasInputFormat = NULL,
    const media_format *hasOutputFormat = NULL,
    char *name = NULL,
    uint64 requireKinds = 0,
    uint64 denyKinds = 0)
```

Queries dormant nodes (those nodes that live in add-ons, rather than in the application) and returns those who match the specified inputs. If *hasInputFormat* isn't **NULL**, the node has to be a BBufferConsumer and have an input format compatible with the format described in *hasInputFormat*. Likewise, if *hasOutputFormat* isn't **NULL**, the node has to be a BBufferProducer that's compatible with the format described in *hasOutputFormat*.

If *name* isn't **NULL**, the node has to have a name that equals *name*, or, if the last character of *name* is an asterisk ("*"), a name whose initial characters match *name* up to, but not including, the asterisk.

The *requireKinds* and *denyKinds* arguments specifiy, respectively, the kinds that must be supported, and the kinds that must not be supported by the returned nodes.

Matching nodes are returned in *outDormantNodeList*. You should pass the size of the *outDormantNodeList* array (the number of elements that the array can hold) in *inOutNumNodes*; when this function returns, the value in *inOutNumNodes* will be changed to the actual number of matching nodes found, unless an error occurs.

Nodes you obtain using `GetDormantNodes()` must be released when you're done using them. To do this, be sure they're stopped and disconnected, then call **`ReleaseNode()`**.

RETURN CODES

- B_OK. No error cloning the node.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be cloned.

GetFileFormatsFor()

```
status_t GetFileFormatsFor(const media_node &fileInterface,
    media_file_format *outFormatList,
    int32 *inOutFormatCount)
```

Given a BFileInterface node in *fileInterface*, returns information about the file formats the file interface can deal with in the array *outFormatList*. On entry, *inOutFormatCount* points to the number of media_file_format structures that can fit in the array specified by *outFormatList*. Upon return, it will contain the actual number of formats returned, unless `GetFileFormatsFor()` returns an error.

This function will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error sending the set mode request.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be cloned.

GetFormatFor()

```
status_t GetFormatFor(const media_output &output,
    media_format *ioFormat, uint32 flags = 0)
status_t GetFormatFor(const media_input &input,
    media_format *ioFormat, uint32 flags = 0)
status_t GetFormatFor(const media_node &node,
    media_format *ioFormat, float quality = B_MEDIA_ANY_QUALITY)
```

`GetFormatFor()` returns the media_format being used by the given object, which may be a media_output, a media_input, or a media_node. Pass in *ioFormat* a pointer to a media_format object to be filled out with the object's format.

The *flags* currently must be zero.

RETURN CODES

- B_OK. No error.
- **B_BAD_VALUE**. You can't pass **NULL** for *ioFormat*.
- **B_MEDIA_BAD_NODE**. The node isn't of the correct type for the given request.
- **B_MEDIA_BAD_SOURCE**. The node's media_source is invalid.
- Messaging errors. An error occurred communicating with the Media Server.

GetFreeInputsFor() see GetConnectedInputsFor()

GetFreeOutputsFor() see GetConnectedOutputsFor()

GetInitialLatencyFor()

```
status_t GetInitialLatencyFor(media_node &producer,
    bigtime_t *outLatency, uint32 *outFlags = NULL)
```

Returns, in *outLatency*, the additional amount of time in microseconds the specified *producer* node requires in order to synchronize to a signal. For example, a TV capture card that's started while the capture is in the middle of a field will have to wait until the next field begins before actually starting to produce buffers.

outFlags is set to the flags returned by the producer. Currently there aren't any flags defined, so this will be returned as zero for now.

RETURN CODES

- B_OK. No error.
- **B_BAD_VALUE**. *outLatency* was specified as **NULL**.

- **B_MEDIA_BAD_NODE**. *producer* isn't a valid node.
- Port errors. An error occurred communicating with the *producer*.

GetInstancesFor()

```
status_t GetInstancesFor(media_addon_id addon, int32 flavor,
    media_node_id *outID, int32 *ioCount = 0)
```

Given the specified *addon* ID and *flavor*, this function fills the *outID* list with up to *ioCount* node IDs that were derived from the specified add-on. If you specify zero for *ioCount*, one node ID will be returned. On return, *ioCount* is changed to indicate how many nodes have been returned in the list.

RETURN CODES

- B_OK. No error.
- Port errors. Communication with the Media Server failed.

GetLatencyFor()

```
status_t GetLatencyFor(const media_node &producer, bigtime_t *outLatency)
```

Reports in *outLatency* the maximum latency found downstream from the specified BBufferProducer, *producer*, given the current connections.

If an error occurs, the value in *outLatency* is unreliable.

This function will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No errors.
- Other errors. Unable to get the latency.

GetLiveNodes()

```
status_t GetLiveNodes(live_node_info *outLiveNodeList, int32 *ioTotalCount,
    const media_format *hasInput = NULL,
    const media_format *hasOutput = NULL,
    const char *name = NULL,
    uint64 nodeKinds = 0)
```

Queries the Media Server for a list of all currently active nodes (whether they're running or not), and fills the array specified by *outLiveNodeList* with information about the nodes. The size of the array should be specified—in terms of how many elements it can contain—by the *ioTotalCount* argument; the actual number of entries in the returned list will be stored in *ioTotalCount* before the call returns.

An active node is a node that is preloaded by the system and is always available for use, as opposed to a dormant node, which resides in an add-on and is only loaded when instantiated using **`InstantiateDormantNode()`**.

You can obtain a more specific result list by specifying one or more of the *hasInput*, *hasOutput*, *name*, and *nodeKinds* arguments. *hasInput* and *hasOutput* let you restrict the resulting list to containing nodes that accept as input (or output) the specified format.

You should always specify 0 for *nodeKinds*; this parameter is currently not used.

RETURN CODES

- B_OK. No errors.
- Other errors. Unable to get the list of live nodes.

GetNodeAttributesFor()

```
ssize_t GetNodeAttributesFor(const media_node &node,
    media_node_attributes *outArray, size_t inMaxCount)
```

Fills the array *outArray* with up to *inMaxCount* attributes of the given *node*. Returns the number of attributes returned. If the result is less than zero, an error occurred.

<<<more on node attributes>>>

GetNodeFor()

```
status_t GetNodeFor(media_node_id nodeID, media_node *clonedNode)
```

Given a *node* specified by node_id, `GetNodeFor()` returns in *clonedNode* a media_node reference to a clone of the node. You can then use the *clonedNode* to query the node for available inputs, outputs, and so forth.

Once your application has finished using the returned node (and it's been stopped and disconnected), you should release it by calling **`ReleaseNode()`**.

RETURN CODES

- B_OK. No error cloning the node.
- BMessage errors. The requested node couldn't be cloned.

GetParameterWebFor()

```
status_t GetParameterWebFor(const media_node & node,
    BParameterWeb **outWeb)
```

Instantiates a BParameterWeb that describes the internal layout of a specific controllable node and stores a pointer to the BParameterWeb in *outWeb*. You can then walk the various BParameters within the web to figure out what there is to control, and to present a user interface to the node's parameters. Delete the web pointed to by *outWeb* when you're done with it.

Note that the `StartControlPanel()` function provides an easy, painless way to automatically present user interface for configuring nodes.

This function will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error obtaining the BParameterWeb.
- `B_MEDIA_BAD_NODE`. The requested node is invalid.

See also: `StartControlPanel()`

GetReadFileFormatListFor() , GetWriteFileFormatListFor()

```
status_t GetReadFileFormatList(const dormant_node_info &inNode,
    media_file_format *outReadFormats,
    int32 inReadCount, int32 *outReadCount)
```

```
status_t GetWriteFileFormatList(const dormant_node_info &inNode,
    media_file_format *outWriteFormats,
    int32 inWriteCount, int32 *outWriteCount)
```

These two functions return lists of file formats that the dormant node described by *inNode* can read or write.

`GetReadFileFormatList()` returns in the array specified by *outReadFormats* a list of file formats the node can read. Specify in *inReadCount* the number of formats that can be held by the *outReadFormats* array. On exit, *outReadCount* indicates how many formats are being returned in the array.

GetWriteFileFormatList() returns in the array specified by *outWriteFormats* a list of file formats the node can write. Specify in *inWriteCount* the number of formats that can be held by the *outWriteFormats* array. On exit, *outWriteCount* indicates how many formats are being returned in the array.

RETURN CODES

- B_OK. The list was returned without error.
- **B_BAD_VALUE**. outReadFormats or outWriteFormats is NULL.
- BMessage errors. An error occurred communicating with the Media Server.

GetRealtimeFlags() , SetRealtimeFlags()

status_t **GetRealtimeFlags(**uint32 **outEnabled**)
status_t **SetRealtimeFlags(**uint32 *inEnabled*)

`GetRealtimeFlags()` returns flags that the Media Server uses to determine whether or not memory needs to be locked down. `SetRealtimeFlags()` sets these flags, and is generally only called by the Media preference application.

Any or all of these flags can be set, in combination.

Constant	Description
B_MEDIA_REALTIME_ALLOCATOR	When set, rtm_alloc() will return locked memory.
B_MEDIA_REALTIME_AUDIO	Audio add-ons in the Media Server are locked in memory, and should lock their thread stacks using **media_init_realtime_thread()**.
B_MEDIA_REALTIME_VIDEO	Video add-ons are locked in memory, and should lock their thread stacks using **media_init_realtime_thread()**.
B_MEDIA_REALTIME_ANYKIND	All Media add-ons are locked in memory, and should lock their thread stacks using **media_init_realtime_thread()**.

See the BMediaNode overview for a discussion of realtime allocation and thread stack locking.

RETURN CODES

- B_OK. No error.
- Other errors. Unable to set or retrieve the realtime flags.

GetRefFor() see SetRefFor()

GetStartLatencyFor()

status_t **GetStartLatencyFor(**const media_node &*timeSource*, bigtime_t **outLatency*)

Reports in *outLatency* the maximum latency found downstream from the time source specified by *timeSource*, given the current connections.

If an error occurs, the value in *outLatency* is unreliable.

This function will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No errors.
- Other errors. Unable to get the latency.

GetSystemTimeSource()

status_t **GetSystemTimeSource(**media_node ***clonedTimeSource*****)**

This function returns, in *clonedTimeSource*, a reference to a clone of the system time source. The system time source is the fallback time source used when no other source is available; its time is derived from the **system_time()** real-time clock. As such, it's quite accurate, but has no relevant relationship to the timing of the hardware devices being used for media input and output. Thus it's not a good choice for a master clock—but it's there if nothing else is available.

By default, new nodes are slaved to the system time source.

RETURN CODES

- B_OK. No error cloning the node.
- **B_NAME_NOT_FOUND**. The time source couldn't be cloned.

GetTimeSource() , MakeTimeSourceFor()

status_t **GetTimeSource(**media_node ***outNode*****)**
BTimeSource ***MakeTimeSourceFor(**media_node &***node*****)**

`GetTimeSource()` returns, in *outNode*, the preferred master clock to which other nodes can be slaved. By slaving all nodes to a single master clock, good synchronization can be ensured.

Typically, the preferred master clock will be the same node as the default audio output (assuming that the audio output node is also a BTimeSource, which should be the case). The sound circuitry's DAC is then used as a timing reference. Although this may be less accurate than the system clock (as defined by the global **system_time()** function), glitch-free audio performance is best ensured by using the audio output to synchronize media operations.

By default, nodes are slaved to the system time source (see **GetSystemTimeSource()** above). Usually you'll want to use this function to obtain a more accurate time source, then slave your nodes to it:

```
media_node timeSource;
roster->GetTimeSource(&media_node);
roster->SetTimeSourceFor(myNode, timeSource.node);
```

This will slave the previously-created node `myNode` to the preferred time source.

`MakeTimeSourceFor()` returns a BTimeSource object corresponding to the specified *node*'s time source. This object can then be used to issue BTimeSource calls to determine and adjust timing issues (for instance, to determine the current performance time). When you're done with the BTimeSource, you should call **`BMediaNode::ReleaseNode()`** on it.

RETURN CODES

- B_OK. No error locating the default input node.
- **`B_NAME_NOT_FOUND`**. The default node couldn't be identified.

- GetVideoInput() see GetAudioInput()
- GetVideoOutput() see GetAudioOutput()
- GetWriteFileFormatListFor() see GetReadFileFormatListFor()

- InstantiateDormantNode()

```
status_t InstantiateDormantNode(const dormant_node_info &inInfo,
    media_node *outNode)
status_t InstantiateDormantNode(const dormant_node_info &inInfo,
    media_node *outNode, uint32 flags)
```

These functions instantiate a node from an add-on, given the information specified in the dormant_node_info structure:

```
struct dormant_node_info {
   dormant_node_info();
   ~dormant_node_info();

   media_addon_id addon;
   int32 flavor_id;
   char name[B_MEDIA_NAME_LENGTH];

private:
   char reserved[128];
};
```

The *addon* field should be filled out to contain the add-on ID of the add-on from which the node should be instantiated, and the *flavor_id* should be the flavor ID number the node

should be instantiated to process. Typically you'll use a function such as `GetDormantNodes()` to find a `dormant_node_info` structure that describes a suitable node.

When you're done using the node, and have stopped and disconnected it, you should always call `ReleaseNode()` to let the Media Server know you're finished with it. This lets the Media Server track whether or not the node's add-on can be unloaded, based on the number of applications still using it.

The difference between these two functions is that the second form lets you specify flags controlling how the node is instantiated. The **`B_FLAVOR_IS_GLOBAL`** flag instantiates the node in the Media Add-on Server's memory space, while the `B_FLAVOR_IS_LOCAL` flag instantiates the node in your application's memory. Using `B_FLAVOR_IS_LOCAL` protects other applications—not to mention the Media Server—from being derailed if the node crashes. Whenever possible, you should instantiate nodes locally. You should only use `B_FLAVOR_IS_GLOBAL` if you need the node to stay around after your application exits.

RETURN CODES

- B_OK. No error instantiating the node.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be instantiated.

MakeTimeSourceFor() see GetTimeSource()

MediaFlags()

static ssize_t **`MediaFlags(`**media_flags *flag*, void **buffer*, size_t *bufferSize*`)`

Asks the Media Server about its support for specific features and capabilities.

The specified *buffer* will be filled with the data indicating the value of the specified flag. If the *buffer* is too small (as indicated by *bufferSize*), only the first *bufferSize* bytes of the result data will be stored in the buffer, but no error will occur.

Constant	Description
`B_MEDIA_FLAGS_VERSION`	Returns the Media Kit version as an int32 value.

If the result is negative, an error occurred, or the Media Server isn't running.

NodeIDFor()

media_node_id **`NodeIDFor(`**port_id *sourceOrDestinationPort*`)`

Given a source or destination port, this function returns the corresponding node's ID number.

PrerollNode()

status_t **PrerollNode(**const media_node &*node*)

Calling `PrerollNode()` sends a preroll message to the specified *node*; the node's **`Preroll()`** hook function will be called. When that hook returns, `PrerollNode()` will also return. A node that's been prerolled should respond very quickly to a **`StartNode()`** call, because the time-consuming setup operations should have been done already by the **`Preroll()`** hook.

While it's not mandatory for an application to call `PrerollNode()` before calling **`StartNode()`**, it's recommended, because doing so may improve real-time performance once the node is started.

This function will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error sending the set mode request.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be cloned.

RegisterNode() , UnregisterNode()

status_t **RegisterNode(**BMediaNode **node***)**
status_t **UnregisterNode(**BMediaNode **node***)**

`RegisterNode()` registers an object of a class derived from BMediaNode with the media roster and assigns it a node_id. This function should be called once a BMediaNode-derived object is fully-constructed and before any attempt is made to connect the node to some other participant in the Media Server.

`RegisterNode()` is called automatically for nodes instantiated from add-ons, but your application will have to call it for any nodes it creates itself.

If you create your own subclass of BMediaNode, its constructor can call `RegisterNode()` itself just before returning (it must be the last thing the constructor does).

`UnregisterNode()` unregisters a node from the Media Server. It's called automatically by the BMediaNode destructor, but it might be convenient to call it sometime before you delete your node instance, depending on your implementation and circumstances.

These functions are generally only used if you're creating your own node class.

RETURN CODES

- B_OK. No error sending the seek request.
- **`B_BAD_VALUE`**. Invalid BMediaNode specified.

ReleaseNode()

```
status_t ReleaseNode(const media_node &node)
```

Releases the specified node, which has previously been obtained by using the `InstantiateDormantNode()`, `GetNodeFor()`, or default node functions (such as `GetVideoNode()` or `GetAudioMixer()`).

RETURN CODES

- B_OK. No error releasing the node.
- BMessage errors. The node couldn't be released.

RollNode()

```
status_t RollNode(const media_node &node, bigtime_t startPerformanceTime,
    bigtime_t stopPerformanceTime,
    bigtime_t atMediaTime = -B_INFINITE_TIMEOUT)
```

Atomically queues a start and stop for the given node. The node will start playing at the performance time indicated by *startPerformanceTime*, and will stop playing at the performance time indicated by *stopPerformanceTime*.

If the *atMediaTime* argument is given, a seek to that media time is also queued.

This function is especially useful for the offline rendering case (the `B_OFFLINE` run mode). It lets you render a certain time range without accidentally going too far; if you queue up a start and stop using `Start()` and `Stop()`, the node may have already rendered past your desired stop time before your `Stop()` call occurs. `RollNode()` avoids that problem.

RETURN CODES

- B_OK. No error.
- `B_MEDIA_BAD_NODE`. The node isn't valid.

Roster() , CurrentRoster()

```
static BMediaRoster *Roster(status_t *outError = NULL)
static BMediaRoster *CurrentRoster(void)
```

`Roster()` returns a pointer to the default BMediaRoster instance, or creates the BMediaRoster instance if it doesn't exist yet, then returns a pointer to it. If you don't want to create the roster if it doesn't already exist, use the `CurrentRoster()` function (it returns `NULL` if there's no roster).

Since `CurrentRoster()` doesn't create a media roster, you obviously must use `Roster()` at least once in your application to create one.

These static member functions should be called by explicit scope, and never by dereference; this is how you get the BMediaRoster through which all other media roster functions are called. For example:

```
BMediaRoster *r = BMediaRoster::Roster();
status_t err = r->GetFreeOutputsFor(some_node, some_array, 3, &n);
```

On return, *outError* is set to **B_OK** if the default BMediaRoster was successfully returned, or a negative error code if something went wrong (for example, if the Media Server isn't running). If *outError* is **NULL**, no error code is returned.

In any case, `Roster()` returns **NULL** if an error occurs.

SeekNode()

```
status_t SeekNode(const media_node &node, bigtime_t newMediaTime,
    bigtime_t atPerformanceTime = 0)
```

Sends the specified *node* a request that it change its playing location to the media time *newMediaTime* once the performance time *atPerformanceTime* is reached.

If the *node* isn't running, the seek request is processed immediately, and the *atPerformanceTime* argument is ignored.

The error returned by this function only indicates whether or not the request was sent successfully; the node may later run into problems trying to perform the seek operation.

If the node is a time source, and you want to operate on the time source aspect of the node (to seek all slaved nodes), you should call **SeekTimeSource()** instead.

RETURN CODES

- B_OK. No error sending the seek request.
- Other errors, depending on the node.

See also: **StartNode()**, **StopNode()**

SeekTimeSource()

```
status_t SeekTimeSource(const media_node &timeSource,
    bigtime_t newPerformanceTime,
    bigtime_t atRealTime)
```

Sends the specified *timeSource* a request that it change the performance time it outputs to its slaved nodes to the time *newPerformanceTime* once the performance time *atRealTime* is reached.

If the *timeSource* isn't running, the seek request is processed immediately, and the *atRealTime* argument is ignored.

The error returned by this function only indicates whether or not the request was sent successfully; the node may later run into problems trying to perform the seek operation.

RETURN CODES

- B_OK. No error sending the seek request.
- Other errors, depending on the node.

See also: `StartTimeSource()`, `StopTimeSource()`

SetAudioInput() , SetVideoInput()

```
status_t SetAudioInput(const media_node &defaultNode)
status_t SetAudioInput(const dormant_node_info &defaultNodeInfo)
```

```
status_t SetVideoInput(const media_node &defaultNode)
status_t SetVideoInput(const dormant_node_info &defaultNodeInfo)
```

These functions set the preferred nodes for audio and video input. If the specified node isn't capable of being the system default, an error will be returned (for example, nodes defined by an application can't be the system default—only nodes defined by Media Kit add-ons can be system defaults).

In general, you shouldn't call these functions unless you're writing software that reimplements the functionality of the BeOS Audio or Video preference panels.

RETURN CODES

- B_OK. No error setting the default input node.
- `B_NAME_NOT_FOUND`. The default node couldn't be changed.

SetAudioOutput() , SetVideoOutput()

```
status_t SetAudioOutput(const media_node &defaultNode)
status_t SetAudioOutput(const dormant_node_info &defaultNodeInfo)
status_t SetAudioOutput(const media_input &inputToOutput)
```

```
status_t SetVideoOutput(const media_node &defaultNode)
status_t SetVideoOutput(const dormant_node_info &defaultNodeInfo)
```

These functions set the preferred nodes for audio and video output. If the specified node isn't capable of being the system default, an error will be returned (for example, nodes defined by an application can't be the system default—only nodes defined by Media Kit add-ons can be system defaults).

In general, you shouldn't call these functions unless you're writing software that reimplements the functionality of the BeOS Audio or Video preference panels.

RETURN CODES

- B_OK. No error setting the default input node.
- **B_NAME_NOT_FOUND**. The default node couldn't be changed.

SetProducerRate()

```
status_t SetProducerRate(const media_node &node,
    int32 numerator, int32 demominator)
```

This function is called to tell the producer to resample the data rate by the specified factor. Specifying a value of 1 (ie, *numerator*/*denominator* = 1) indicates that the data should be output at the same playback rate that it comes into the node at. The format of the data should be unchanged.

Nodes are not required to support this mechanism for controlling their data rate, so this call may have no effect.

RETURN CODES

- B_OK. No error.
- Other errors. Returned by **BBufferProducer::SetPlayRate()**.

See also: **BBufferProducer::SetPlayRate()**

SetProducerRunModeDelay()

```
status_t SetProducerRunModeDelay(const media_node &node,
    bigtime_t delay,
    BMediaNode::run_mode mode = B_RECORDING)
```

Sets the run mode for the given producer *node* to *mode*. Also sets the specified *delay* to be added to each buffer sent by the producer *node*. This function should only be called for **B_RECORDING** mode; it's provided to compensate when you connect a node that's in recording mode to a node that isn't.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_NODE**. The *node* is invalid.
- Port errors. An error occurred communicating with the Media Server.

SetRealtimeFlags() see GetRealtimeFlags()

SetRefFor() , GetRefFor()

```
status_t SetRefFor(const media_node &fileInterface,
    entry_ref &file,
    bool createAndTruncate,
    bigtime_t *outDuration)
```

```
status_t GetRefFor(const media_node &fileInterface,
    entry_ref *outFile,
    BMimeType *outMimeType = NULL)
```

`SetRefFor()` tells the BFileInterface *fileInterface* to work on the file whose entry_ref is specified by *file*. If *createAndTruncate* is **true**, any previous file with that reference is deleted and the file will be prepared for new output. If *createAndTruncate* is **false**, *outDuration* will, on return, contain the duration of the performance data found in the file.

`GetRefFor()` fills out the specified entry_ref, *outFile*, to reference the file with which the specified *fileInterface* node is working. If *outMimeType* isn't **NULL**, it'll contain a BMimeType object describing the file's type, unless an error occurs.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_NODE**. The specified node doesn't exist, or isn't a file interface.

SetRunModeNode()

```
status_t SetRunModeNode(const media_node &node,
    BMediaNode::run_mode newMode)
```

Sends the specified *node* a request that it change its policy for handling situations where it falls behind during real-time processing.

The error returned by this function only indicates whether or not the request was sent successfully.

RETURN CODES

- B_OK. No error sending the set mode request.
- **B_BAD_NODE**. The specified node is invalid.

SetTimeSourceFor()

```
status_t SetTimeSourceFor(media_node_id node,
    media_node_id timeSource)
```

Tells the specified node to slave its timing to *timeSource*. Once this is done, the node will receive its notion of the passage of time from *timeSource*. As such, it will pause whenever *timeSource* is stopped, and so forth.

By default, nodes are slaved to the system time source, so you only need to call this function if you need to slave a node to a different time source.

The *node* will take whatever precautions are necessary to remain faithful to the notion of time presented by *timeSource* without causing glitches in the presentation of its media. For example, if a sound card node has a DAC that drifts from *timeSource*, it might try to fix the problem by varying the sampling rate slightly, or by dropping or doubling buffers occasionally. This is why you should usually use the preferred time source—rather than the system time source—as your master time source. The preferred time source will usually be derived directly from the DAC being used to produce the media output.

RETURN CODES

- B_OK. No error.
- BMessage errors. Unable to set the time source.

SetVideoInput() see SetAudioInput()

SetVideoOutput() see SetAudioOutput()

SniffRef() , SniffRefFor()

```
status_t SniffRef(const entry_ref &file,
    uint64 requireNodeKinds,
    dormant_node_info *outNode,
    BMimeType outMimeType = NULL)
```

```
status_t SniffRefFor(const media_node &fileInterface,
```

```
    const entry_ref &file,
    BMimeType outMimeType,
    float *outCapability)
```

`SniffRef()` asks all BMediaAddOn instances that satisfy the *requireNodeKinds* restraint to identify the *file*. The *requireNodeKinds* argument should contain flags composited from the **`node_kind`** constants.

The node that returns the greatest *outCapability* value will be chosen, and a reference to it put in *outNode*. The MIME type of the file will be put into *outMimeType*.

In simpler terms: `SniffRef()` returns the node that can best handle the media data in the specified file.

`SniffRefFor()`, on the other hand, asks the specified *fileInterface* node to examine the *file*. If the node recognizes the file, the MIME type of the file is stored in the buffer *outMimeType*, and the node's capability to handle the file is returned in *outCapability*.

If the node doesn't recognize the file, an error is returned. If the node recognizes the file format but finds no recognizable data within the file, *outCapability* is set to 0.0 and no error is returned.

In either case, the higher the *outCapability* value returned, the more appropriate the node is for handling the media data in the file.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error sending the set mode request.
- **`B_MEDIA_BAD_NODE`**. The specified node is invalid, or isn't a file interface.

SniffRefFor() see SniffRef()

StartControlPanel()

```
status_t StartControlPanel(const media_node node,
    BMessenger *outMessenger = NULL)
```

Tells the specified *node* to start its custom control panel, which is started outside your application. There's no way to tell when the user has closed the control panel, other than by indirectly detecting possible changes to the node, such as a renegotiation of the format of data being output by the node.

If a BMessenger is provided as input to `StartControlPanel()`, the function returns in *outMessenger* a BMessneger that can be used to communicate with the control panel.

These functions will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error starting the control panel.
- **B_MEDIA_BAD_NODE**. The specified node is invalid.

StartNode() , StopNode()

```
status_t StartNode(const media_node &node, bigtime_t atPerformanceTime)
status_t StopNode(const media_node &node, bigtime_t atPerformanceTime,
    bool immediate = false)
```

`StartNode()` sends the specified *node* a request to start streaming data at the performance time specified by the *atPerformanceTime* argument, according to that node's time source.

By default, nodes are in a stopped state upon creation, so you have to call `StartNode()` once you have a reference to it before anything will happen. Starting a node that's already running has no effect.

`StopNode()` sends node a request to stop streaming data once the specified performance time *atPerformanceTime* is reached, according to that node's time source. Stopping a node that's already stopped has no effect. If *immediate* is **true**, the node is instructed to stop immediately and the *atPerformanceTime* argument is ignored; if *immediate* is **false**, the node is stopped at the specified performance time.

In either case, the requested change will occur at the time specified by *atPerformanceTime*.

If the node is a time source, and you want to operate on the time source aspect of the node (to start or stop all slaved nodes), you should call **SeekTimeSource()** instead.

The error returned by these functions only indicates whether or not the request was sent successfully; the node may later run into problems trying to start or stop its media and you won't know it based on the result of these functions.

`StopNode()` will deadlock if called from a node's control thread or while the control thread is blocked.

RETURN CODES

- B_OK. No error sending the start or stop request.

- Other errors, depending on the node.

See also: `SeekNode()`

StartTimeSource() , StopTimeSource()

```
status_t StartTimeSource(const media_node &timeSource, bigtime_t atRealTime)
status_t StopTimeSource(const media_node &timeSource, bigtime_t atRealTime,
    bool immediate = false)
```

`StartTimeSource()` sends the specified *timeSource* a request to start running at the real time specified by the *atRealTime* argument.

`StopTimeSource()` sends node a request to stop the specified *timeSource* once the specified real time *atRealTime* is reached. Stopping a time source that's already stopped has no effect. If *immediate* is **true**, the time source is instructed to stop immediately and the *atRealTime* argument is ignored; if *immediate* is **false**, the time source is stopped at the specified real time.

In either case, the requested change will occur at the time specified by *atRealTime*.

The error returned by these functions only indicates whether or not the request was sent successfully; the node may later run into problems trying to start or stop and you won't know it based on the result of these functions.

RETURN CODES

- B_OK. No error sending the start or stop request.
- Other errors, depending on the node.

See also: `SeekTimeSource()`

StartWatching() , StopWatching()

```
status_t StartWatching(const BMessenger &notifyHandler)
status_t StartWatching(const BMessenger &notifyHandler,
    int32 notificationType)
status_t StartWatching(const BMessenger &notifyHandler,
    const media_node &node,
    int32 notificationType)
```

```
status_t StopWatching(const BMessenger &notifyHandler)
status_t StopWatching(const BMessenger &notifyHandler,
    int32 notificationType)
status_t StopWatching(const BMessenger &notifyHandler,
    const media_node &node,
    int32 notificationType)
```

`StartWatching()` registers the specified BHandler or BLooper as a recipient of notification messages from the Media Server. `StopWatching()` cancels this registration so that no further notifications will be sent.

If you're only interested in a particular notification type, you can specify that code in the *notificationType* argument. If you don't specify a notification type, **`B_MEDIA_WILDCARD`** is assumed; this matches all notification types. You can also specify that you want to watch a specific *node*; if you don't specify a node, you'll receive notifications for all nodes.

Events are sent to registered BHandlers and BLoopers when certain events happen, such as nodes being created or deleted, or connections being made or broken.

See "Notification Messages" for a list of the notifications you can receive, and the formats of the corresponding messages.

Although the Media Server will automatically cancel notifications to BHandlers and BLoopers that go away without explicitly calling `StopWatching()`, this detection is expensive and may briefly interrupt the media system, so you should always call `StopWatching()` before allowing a BHandler or BLooper to go away.

RETURN CODES

- B_OK. No error cloning the node.
- **`B_NAME_NOT_FOUND`**. The requested node couldn't be cloned.

StopNode() see StartNode()

StopWatching() see StartWatching()

SyncToNode()

```
status_t SyncNode(const media_node &node, bigtime_t atPerformanceTime,
    bigtime_t timeout = B_INFINITE_TIMEOUT)
```

If you want to detect the arrival of a specific performance time on a given node, you can do that by calling `SyncToNode()`. Specify the node you want to monitor in *node*, and the time you want to be notified of in *atPerformanceTime*. You can, optionally, specify a *timeout*; if the sync hasn't occurred in *timeout* microseconds, the request will time out.

This function doesn't return until either the specified performance time arrives, or the sync operation times out.

RETURN CODES

- B_OK. All's well.
- **`B_MEDIA_BAD_NODE`**. The specified *node* is invalid.

- `B_TIMED_OUT`. The request timed out.
- Port errors. An error occurred communicating with the Media Server.

Constants

connect_flags

Declared in: be/media/MediaRoster.h

Constant	Description
`B_CONNECT_MUTED`	The connection should be muted on creation.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaRoster Function Summary

Function Summary

Function	Description
AudioBufferSizeFor()	Determines the preferred audio buffer size for the described audio format
Connect()	Establishes a connection between a media source and a media destination
CurrentRoster()	Returns the default BMediaRoster instance
Disconnect()	Terminates the specified connection
GetAllInputsFor()	Obtains a list of all of a node's inputs
GetAllOutputsFor()	Obtains a list of all a node's outputs
GetAudioInput()	Returns the default node for audio or video input
GetAudioOutput()	Returns the default node for audio or video output, or for auxio mixing
GetConnectedInputsFor()	Obtains a list of a node's inputs that are connected to an output
GetConnectedOutputsFor()	Obtains a list of a node's outputs that are connected to an input

GetDormantFlavorInfoFor()	Returns flavor information for a dormant node
GetDormantNodes()	Obtains a list of dormant nodes (nodes that live in add-ons)
GetFileFormatsFor()	Returns a list of file formats supported by a file interface node
GetFormatFor()	Returns a media_format indicating the format for the given media_input, media_output, or media_node
GetFreeInputsFor()	Obtains a list of a node's inputs that aren't in use
GetFreeOutputsFor()	Obtains a list of a node's outputs that aren't in use
GetInitialLatencyFor()	Returns the amount of time required to start the specified node
GetInstancesFor()	Returns a list of media_node_ids for all nodes derived from a given add-on and flavor
GetLatencyFor()	Returns the maximum latency downstream from a producer
GetLiveNodes()	Obtains a list of all active nodes
GetNodeAttributesFor()	Returns the specified node's attributes

GetNodeFor()	Returns a media_node for a node given by a node ID
GetParameterWebFor()	Returns a BParameterWeb for a controllable node
GetReadFileFormatListFor()	Returns a list of file formats that can be read by the specified dormant node
GetRealtimeFlags()	Returns the real time flags
GetRefFor()	Returns the file that a node is working with
GetStartLatencyFor()	Returns the amount of time it takes to start the specified node
GetSystemTimeSource()	Returns a clone of the system time source
GetTimeSource()	Returns the preferred master clock for synchronization of nodes
GetWriteFileFormatListFor()	Returns a list of file formats that can be written by the specified dormant node
InstantiateDormantNode()	Instantiates a node that lives in an add-on
MakeTimeSourceFor()	Returns a BTimeSource for the specified time source media_node
MediaFlags()	Returns information about Media Kit capabilities
BMediaRoster()	Constructor

NodeIDFor()	Returns the node ID that matches the specified port
PrerollNode()	Asks that the specified node prepare to be started
RegisterNode()	Registers a node with the Media Server
ReleaseNode()	Releases a node you've been using
RollNode()	Atomically queues a start and a stop request for a node in one call
Roster()	Returns the default BMediaRoster instance, creating it if it doesn't exist
SeekNode()	Requests that a node change its playing or recording location
SeekTimeSource()	Requests that a time source shift in performance time
SetAudioInput()	Sets the default node for audio or video input
SetAudioOutput()	Sets the default node for audio or video output
SetProducerRate()	Tells a producer to resample its data rate by a specified factor
SetProducerRunModeDelay()	Sets a producer's run mode and sets up a delay to be

	added to each buffer being sent
SetRealtimeFlags()	Sets the real time flags
SetRefFor()	Sets the file with which a node will work
SetRunModeNode()	Specifies how a node should deal with falling behind while doing real-time processing
SetTimeSourceFor()	Slaves a node to a specified time source
SniffRef()	Ask all matching nodes to identify a file
SniffRefFor()	Ask a specific node to identify a file
StartControlPanel()	Starts a node's custom control panel
StartNode()	Requests that a node start streaming data
StartTimeSource()	Requests that a time source start
StartWatching()	Enables notifications when certain Media Server events occur
StopNode()	Requests that a node stop streaming data
StopTimeSource()	Requests that a time source stop
StopWatching()	Cancels notifications of Media Server events

SyncToNode()	Asks the specified node to notify you when the indicated performance time arrives
UnregisterNode()	Unregisters a node with the Media Server

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSoundPlayer

Derived from: (none)
Declared in: be/media/SoundPlayer.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BSoundPlayer class plays sound, either from BSound objects or by directly filling audio buffers with data provided by a hook function you specify. A single BSoundPlayer can play multiple sounds at a time.

BSoundPlayer takes care of all the nitty-gritty details of instantiating the necessary sound player node and managing the time source. All you have to do is start up a BSoundPlayer and feed it sounds to play.

For simple audio playback, the sounds are specified using the BSound class. However, if you need to perform on-the-fly generation or alteration of sounds, you can implement a hook function that will be called for each audio buffer passed through the BSoundPlayer's playback node.

Using BSoundPlayer

Once you've instantiated a BSoundPlayer object, you need to start it up before you can actually play sounds with it. This instantiates the sound player node, attaches it to an appropriate time source, and makes sure the time source is running. This is done by calling the `Start()` function.

When you're done using the BSoundPlayer, you can delete it if you don't plan to use it again, or, if you want to keep it around for reuse, you can just `Stop()` it. This deletes the sound node and cleans up the sounds.

In general, the process of using BSoundPlayer looks like this:

```
BSoundPlayer player;
player.Start();

/* use the BSoundPlayer here */

player.Stop();
```

You can find out the current time of the time source to which sounds are being synchronized by calling the `CurrentTime()` function.

By default, the audio format used by a BSoundPlayer is **`BSoundPlayer::B_AUDIO_FLOAT`**, which is to say that the audio is in floating-point format, where each sample ranges from -1.0 to 1.0. The Media Kit uses floating-point audio internally, so using floating-point audio whenever you can will improve your application's performance by cutting down on format conversions. However, if you want to use another format, you may do so by specifying a media_raw_audio_format when you instantiate the BSoundPlayer object.

In the current release of BeOS, all BSounds playing through a single BSoundPlayer must have the same sampling rate. This restriction may be lifted in the future. For the time being, just instantiate one BSoundPlayer for each sampling rate you need to support. In addition, only the **`BSoundPlayer::B_AUDIO_FLOAT`** audio format is supported.

Sound streams may only contain 1 or 2 channels (either mono or stereo, in other words). There's no support yet for multichannel sound.

BSoundPlayer can only play sounds in the native byte order.

The audio mixer performs best if you don't specify a particular buffer size.

Playing Sound

A BSoundPlayer object plays sounds asynchronously, and can play as many sounds as you like. You start playback of a sound by passing a BSound pointer to `StartPlaying()`:

```
play_id id = player->StartPlaying(sound);
```

StartPlaying() returns an ID number you can use to refer to the sound throughout its existence; other BSoundPlayer functions require this ID as an argument.

You can, optionally, specify a time at which to begin playback:

```
play_id = player->StartPlaying(sound, 100000);
```

This starts playback at the performance time 100000 microseconds.

You can determine whether or not a sound is still playing by calling **IsPlaying()**:

```
if (player->IsPlaying(id)) {
   /* the sound is playing */
}
```

You can stop the sound by calling **StopSound()**:

```
player->StopSound(id);
```

And the **WaitForSound()** function waits until the specified sound is finished playing:

```
play_id id = player->StartPlaying(sound);
player->WaitForSound(id);
```

And you can control the volume as well. Use the **Volume()** function to determine the current volume, and **SetVolume()** to change the volume. The volume is a floating-point number, and ranges from 0.0 (silence) to 1.0 (full volume). Here's a code snippet that halves the current volume:

```
SetVolume(Volume()/2.0);
```

Example: Let's Play Around

Playing sound files using BSoundPlayer is very easy. Here's a function that will play a specified sound, waiting until the sound is finished playing before it returns:

```
void playsound(char *path) {
   BSound *sound;
   BSoundPlayer player;
   entry_ref ref;
   BEntry entry(path, true);
   BSoundPlayer::play_id id;

   if (entry.InitCheck()) == B_OK) {
      if (entry.GetRef(&ref) == B_OK) {
```

```
         sound = new BSound(&ref);
         if (sound->InitCheck() == B_OK) {
            player.Start();
            player.SetVolume(1.0);
            id = player.StartPlaying(sound);
            sound->ReleaseRef();
            player.WaitForSound(id);
         }
      }
   }
}
```

An entry_ref is obtained for the specified sound file, then a BSound object is instantiated given that entry_ref. A BSoundPlayer object is started, the volume is set to 1.0 (full volume), and then the sound playback is started by calling **StartPlaying()**. The sound's reference count is decremented, and finally BSoundPlayer's **WaitForSound()** call is used to wait until playback is complete.

Advanced Playback

If you have more advanced playback needs—for example, you need to play sounds that are being generated on-the-fly, or if you need to filter the sound you're playing.

If you specify a play buffer handler function, either when instantiating the BSoundPlayer object or by calling **SetCallbacks()** or **SetBufferPlayer()**, that function will be called once for each buffer that passes through the BSoundPlayer's sound playing node. Your play buffer handler can then fill the buffer with whatever data you wish.

If you specify a play buffer function, BSound playback won't work. If you need to do both, you must use multiple BSoundPlayer objects.

The following code sets up a BSoundPlayer that will play a triangle wave.

```
typedef struct cookie_record {
   float value;
   float direction;
} cookie_record;

...
cookie_record cookie;

cookie.value = 0.0;
cookie.direction = 1.0;

BSoundPlayer player("wave_player", BufferProc, NULL, &cookie);
player.Start();
player.SetHasData(true);
```

```
...
player.Stop();
```

This code establishes a record, cookie, that contains information the play buffer function will need to track, and creates a BSoundPlayer named "wave_player" that will use a function called `BufferProc()` to play sound, and uses the `cookie` we've created.

Then the player is started, and **`SetHasData()`** is called to let the sound player node know that there's data to be played. This will cause the play buffer function to start being called.

Once playback is over, the **`Stop()`** function is called to stop playback.

The `BufferProc()` function looks like this:

```
void BufferProc(void *theCookie, void *buffer, size_t size,
            const media_raw_audio_format &format) {
   size_t i, j;
   float *buf = (float *) buffer;
   size_t float_size = size/4;
   uint32 channel_count = format.channel_count;
   cookie_record *cookie = (cookie_record *) theCookie;

   // We're going to be cheap and only work for floating-point audio

   if (format.format != media_raw_audio_format::B_AUDIO_FLOAT) {
      return;
   }

   // Now fill the buffer with sound!

   for (i=0; i<float_size; i+=channel_count) {
      for (j=0; j<channel_count; j++) {
         buf[i+j] = cookie->value;
      }
      if ((cookie->direction == 1.0) && (cookie->value >= 1.0)) {
         cookie->direction = -1.0;
      }
      else if ((cookie->direction == -1.0) && (cookie->value <= -1.0)) {
         cookie->direction = 1.0;
      }
      cookie->value += cookie->direction*(1.0/64.0);
   }
}
```

This example play buffer function generates a triangle wave, ramping the wave up and down from 1.0 to -1.0 and back, over and over again, 1/64th at a time. The next value to store in the buffer and the direction in which the value is changing are kept in the cookie's fields.

The buffers your play buffer function receives are empty. Do with them as you please (or do nothing at all).

Hook Function

- `HasData()`

Constructor and Destructor

BSoundPlayer()

```
BSoundPlayer(const char *name = NULL,
    void (*PlayBuffer)(void *, void *buffer, size_t size, const media_raw_audio_format &format) = N
    void (*Notifier)(void *, sound_player_notification what, ...) = NULL,
    void *cookie = NULL)
BSoundPlayer(const media_raw_audio_format * format,
    const char *name = NULL,
    void (*PlayBuffer)(void *, void * buffer, size_t size, const media_raw_audio_format & format) =
    void (*Notifier)(void *, sound_player_notification what, ...) = NULL,
    void *cookie = NULL)
BSoundPlayer(const media_node & toNode,
    const media_multi_audio_format *format = NULL,
    const char *name = NULL,
    const media_input *input = NULL,
    void (*PlayBuffer)(void *, void * buffer, size_t size, const media_raw_audio_format & format) =
    void (*Notifier)(void *, sound_player_notification what, ...) = NULL,
    void *cookie = NULL)
```

Initializes the BSoundPlayer object. The *name* argument specifies the name to be assigned to the sound player node (if you specify **NULL**, a generic name will be assigned).

The *PlayBuffer* argument specifies a pointer to a member function that processes data and inserts it into buffers for playback; specify **NULL** if you want to use the BSoundPlayer for playing BSounds. The parameters to the *PlayBuffer* function are (in order):

- Pointer to the cookie.
- Pointer to the buffer to play.
- Size of the buffer.
- Format of the audio data in the buffer.

The *Notifier* parameter specifies a pointer to a member function that receives notifications when events of interest occur, such as playback starting or stopping. Specify **NULL** to use the default notification handler. There are three possible notifications:

- **B_STARTED** and **B_STOPPED** indicate that the BSoundPlayer was started or stopped. These receive a pointer to the BSoundPlayer object as an argument.
- **B_SOUND_DONE** indicates that a sound has finished playing. In this case, there are two arguments: the first is a play_id indicating which sound finished playing, and the other is a boolean which is **true** if the sound played at all, and **false** if the sound never played.

If the callback handlers are members of a class, they must be static members.

The *cookie* parameter is a pointer that you can use for your own purposes; it's most useful if you're using a custom *PlayBuffer* or *Notifier*.

The second form of the constructor lets you specify in the *format* argument the format of the audio that the BSoundPlayer will perform. Since BSoundPlayer can only play raw sound formats, this is specified using the `media_raw_audio_format` structure.

The third form of the constructor lets you specify a node through which the sound should be played, and also uses a media_multi_audio_format to specify the sound's format, instead of the older media_raw_audio_format.

You should call `InitCheck()` before using your BSoundPlayer object; this will let you determine whether or not the object was successfully constructed. One situation in which `InitCheck()` might indicate an error is if the user doesn't have a sound card installed.

~BSoundPlayer()

```
~BSoundPlayer()
```

Stops playback, if sound is playing, releases references to any BSound objects that are in use by the BSoundPlayer, and frees all memory used by the BSoundPlayer.

Member Functions

BufferPlayer() , SetBufferPlayer()

BufferPlayerFunc **BufferPlayer**(void) const
void **SetBufferPlayer**(void (**PlayBuffer*)(void *, void *buffer, size_t size, const media_raw_audio_ &*format*))

`BufferPlayer()` returns a pointer to the current play buffer function, or **`NULL`** if the default player is in use.

`SetBufferPlayer()` lets you change the play buffer function.

Cookie() , SetCookie()

void ***Cookie**(void) const
void **SetCookie**(void **cookie*)

`Cookie()` returns the current cookie assigned to the BSoundPlayer.

`SetCookie()` lets you change the cookie assigned to the BSoundPlayer.

CurrentTime() , PerformanceTime()

bigtime_t **CurrentTime**(void)
bigtime_t **PerformanceTime**(void)

`CurrentTime()` returns the current media time, and `PerformanceTime()` returns the current performance time of the sound player node being used by the BSoundPlayer.

`PerformanceTime()` will return **`B_ERROR`** if the BSoundPlayer object hasn't been properly initialized.

EventNotifier() , SetEventNotifier()

EventNotifierFunc **EventNotifier**(void) const
void **SetNotifier**(void (**Notifier*)(void *, sound_player_notification what, ...))

`EventNotifier()` returns a pointer to the current event notification handler function, or **`NULL`** if the default player is in use.

`SetNotifier()` lets you change the event notification handler function.

Format()

```
media_raw_audio_format Format(void) const
```

Returns the BSoundPlayer's format. Since the sound is always a raw sound format, the `media_raw_audio_format` structure is used.

GetVolumeInfo()

```
status_t GetVolumeInfo(media_node *outNode,
    int32 *outParameter,
    float *outMinDB, float *outMaxDB) const
```

Returns information about the BSoundPlayer's volume control. Pass pointers to variables to be filled (**NULL** is not permitted), and on return these values will be set to describe the player as follows:

- *outNode* is the node to which the BSoundPlayer's audio output will be directed.
- *outParameter* is the parameter ID for the volume control.
- *outMinDB* is the minimum volume in decibels.
- *outMaxDB* is the maximum volume in decibels.

HasData() , SetHasData()

```
bool HasData(void)
void SetHasData(bool hasData)
```

`HasData()` returns **true** if there's sound queued for playback, or **false** otherwise.

`SetHasData()` specifies whether or not there's sound scheduled for playback.

The purpose of these functions is to optimize the BSoundPlayer; if there's no data queued for playback, the sound player node is told this, which lets it optimize its performance. If you're using a buffer player function, you must use `SetHasData()` to indicate that there's data to play:

```
SetHasData(true);
```

InitCheck()

```
status_t InitCheck(void)
```

Returns the status code resulting from constructing the BSoundPlayer object. You should call this after constructing the object; if the returned value is anything other than **B_OK**, you shouldn't use the object.

- IsPlaying() see StartPlaying()

- Latency()

```
bigtime_t Latency(void)
```

Returns the BSoundPlayer's latency.

- PerformanceTime() see CurrentTime()
- Set see Cookie()
- Set see BufferPlayer()

- SetCallbacks()

```
void SetCallbacks(void (*PlayBuffer)(void *, void *buffer, size_t size, const media_raw_audio_for
&format) = NULL,
    void (*Notifier)(void *, sound_player_notification what, ...) = NULL,
    void *cookie = NULL)
```

Sets the play buffer handler function, the event notification handler function, and the cookie all in one atomic operation.

- Set see Cookie()
- Set see HasData()

- SetInitError()

protected:

```
void SetInitError(status_t inError)
```

Sets the status code that will be returned by **InitCheck()**.

- SetNotifier() see EventNotifier()

- SetSoundVolume()

status_t **SetSoundVolume(**play_id *sound*, float *volume***)**

Sets the *volume* (from 0.0 to 1.0) of the specified *sound*.

- Set see Volume()
- Set see VolumeDB()

Start() , Stop()

status_t **Start(**void**)**
void **Stop(**bool *block* = true, bool *flush* = true**)**

`Start()` activates the BSoundPlayer by starting the time source and the sound player node. The **`B_STARTED`** notification is sent to the BSoundPlayer's notification handler.

`Stop()` deactivates the BSoundPlayer by stopping the player node (if *block* is **`true`**, the `Stop()` function blocks until the node is stopped). If *flush* is **`true`**, the queued sounds are all deleted from memory.

While the BSoundPlayer is running, the play buffer function (if you've specified one) will be called for each buffer that passes through the BSoundPlayer's playback node. This hook function can be used to implement code that performs more advanced playback of sound, such as sound that's generated on-the-fly, or is filtered before playback.

RETURN CODES

- B_OK. The sound playback was started.
- Other errors. As returned by the sound player's **`StartNode()`** function.

StartPlaying() , StopPlaying() , WaitForSound() , IsPlaying()

play_id **StartPlaying(**BSound **sound*, bigtime_t *atTime* = 0**)** play_id **StartPlaying(**BSound **sound*, bigtime_t *atTime*, float *withVolume***)**
status_t **StopPlaying(**play_id *id***)**
status_t **WaitForSound(**play_id *id***)**
bool **IsPlaying(**play_id *id***)**

`StartPlaying()` schedules the specified BSound to begin playback at the performance time specified by *atTime*; if *atTime* is 0, the sound begins playing immediately (or as soon as **`Start()`** is called, if the BSoundPlayer hasn't been started yet). The play_id returned by this function is used to identify the sound later. If it's negative, an error occurred (see the list below for

possible values). The second form of `StartPlaying()` lets you specify a volume at which the sound should play.

`StopPlaying()` stops playing the sound specified by the given *id*.

`WaitForSound()` waits until the specified sound stops playing.

`IsPlaying()` returns **`true`** if the specified sound is playing; otherwise, it returns **`false`**.

RETURN CODES

- B_OK. No error.
- **`B_MEDIA_BAD_FORMAT`**. The audio isn't in a supported format (`StartPlaying()`).
- **`B_BAD_VALUE`**. The specified id doesn't exist (`WaitForSound()` and `IsPlaying()`).

StopPlaying() see StartPlaying()

Volume() , SetVolume()

float **`Volume(`**void**`)`**
void **`SetVolume(`**float *newVolume*`)`**

`Volume()` returns the current playback volume.

`SetVolume()` changes the playback volume.

This volume is the overall volume of all the sounds being played by the BSoundPlayer. To control the volumes of individual sounds, use the **`SetSoundVolume()`** function.

The volume can range from 0.0 to 1.0, where 0.0 is silent and 1.0 is maximum loudness.

If you'd rather handle the volume using decibels instead of the percentage range, you can use the **`VolumeDB()`** and **`SetVolumeDB()`** functions.

See also: **`VolumeDB()`** and **`SetVolumeDB()`**

VolumeDB() , SetVolumeDB()

float **`VolumeDB(`**void**`)`**
void **`SetVolumeDV(`**float *newVolumeDB*`)`**

`VolumeDB()` returns the current playback volume in decibels.

`SetVolumeDB()` sets the playback volume, in decibels.

This volume is the overall volume of all the sounds being played by the BSoundPlayer. To control the volumes of individual sounds, use the `SetSoundVolume()` function.

The possible range of volumes can be obtained by calling `GetVolumeInfo()`.

If you'd rather handle the volume using a percentage range instead of decibels, you can use the `Volume()` and `SetVolume()` functions.

See also: `Volume()` and `SetVolume()`

WaitForSound() see StartPlaying()

Constants

sound_player_notification

Constant	Meaning
B_STARTED	The BSoundPlayer has been started via the **Start()** function.
B_STOPPED	The BSoundPlayer's **Stop()** function has been called.
B_SOUND_DONE	A sound has finished playing.

These constants are passed to event notification handler functions to indicate what sort of interesting event has occurred.

Defined Types

play_id

```
typedef int32 play_id;
```

Identifies a particular sound that's being played by the BSoundPlayer; `StartPlaying()` returns values of this type.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSoundPlayer Function Summary

Function Summary

Function	Description
BufferPlayer()	Returns a pointer to the buffer player function
Cookie()	Returns the cookie
CurrentTime()	Returns the current media time
EventNotifier()	Returns a pointer to the notification handler function
Format()	Returns the BSoundPlayer's format
GetVolumeInfo()	Returns information about the BSoundPlayer's volume control
HasData()	Reports whether or not there's sound queued for playback
InitCheck()	Indicates the result of the constructor
IsPlaying()	Checks to see if a particular sound is playing
Latency()	Returns the BSoundPlayer's latency
PerformanceTime()	Returns the current performance time
SetBufferPlayer()	Sets the buffer player function
SetCallbacks()	Sets the callbacks and the cookie at once, atomically
SetCookie()	Sets the cookie
SetEventNotifier()	Sets the notification handler function
SetHasData()	Specifies whether or not there's sound scheduled for playback

SetInitError()	Sets the value returned by InitCheck()
SetSoundVolume()	Sets the volume of a specific sound
SetVolume()	Sets the BSoundPlayer's volume as a percentage from 0.0 to 1.0
SetVolumeDB()	Sets the BSoundPlayer's volume in decibels
BSoundPlayer()	Constructor
~BSoundPlayer()	Destructor
Start()	Activates the BSoundPlayer
StartPlaying()	Starts playing a sound
Stop()	Deactivates the BSoundPlayer
StopPlaying()	Stops a sound that's already playing
Volume()	Returns the BSoundPlayer's volume as a percentage from 0.0 to 1.0
VolumeDB()	Returns the BSoundPlayer's volume in decibels
WaitForSound()	Waits for a particular sound to finish playing

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFileInterface

Derived from: BMediaNode
Declared in: be/media/FileInterface.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A node that can read data from, or write data to, a disk file should derive from BFileInterface in order to allow applications to easily specify what file the node should work with. The node will then be called upon by the Media Server to try to identify, and possibly work with, unknown files.

Your node can't just derive from BFileInterface; it must also derive from at least one of BBufferConsumer and BBufferProducer, depending on its needs.

Hook Functions

- DisposeFileFormatCookie()
- GetDuration()
- GetNextFileFormat()
- GetRef()
- HandleMessage()
- SetRef()
- SniffRef()

Constructor

BFileInterface()

protected:

```
BFileInterface(void)
```

Constructor.

Member Functions

DisposeFileFormatCookie()

protected:

```
virtual void DisposeFileFormatCookie(int32 cookie) = 0
```

This function must be implemented to dispose of a *cookie* used for iterating over file formats your node supports. If the cookie is a reference to a data block, you should free it; otherwise, you can just return without doing anything.

HandleMessage()

protected:

```
virtual status_t HandleMessage(int32 message, const void *data, size_t size)
```

Implement this function to handle messages that arrive on your control port; see `BMediaNode::HandleMessage()` for details.

GetDuration()

protected:

```
virtual status_t GetDuration(bigtime_t *outDuration) = 0
```

Implement this function to return in *outDuration* the duration, in microseconds of the media data in the file currently being worked on.

If there is no file in use, or the file isn't valid, return a negative error code; otherwise, return `B_OK`.

GetNextFileFormat()

protected:

```
virtual status_t GetNextFileFormat(int32 *cookie,
    media_file_format *outFormat) = 0
```

The first time this function is called, *cookie* will be 0. Return information about the first file format you support in *outFormat* and set cookie to some meaningful (non-zero) value that you can use to track where you are in the list of formats you support, then return `B_OK`.

On successive calls to `GetNextFileFormat()`, you should return successive file format information and change *cookie* as necessary to remember where you are in the list. Return **B_OK** each time you successfully return information about a file format.

Once you run out of formats, return **B_ERROR**.

GetRef() , SetRef()

protected:

```
virtual status_t GetRef(const entry_ref *outRef, BMimeType *outMimeType) = 0
virtual status_t SetRef(const entry_ref &file, bool create,
    bigtime_t *outDuration) = 0
```

Implement `GetRef()` function to return, in *outRef*, the entry_ref of the file with which the node is currently working. You should also return the MIME type of the file in *outMimeType*.

When an application or other client wants your node to use a specific file, the `SetRef()` function will be called. The file to be used is specified by *file*, which may or may not exist. If *create* is **true**, you should create a new file (erasing an existing file, if one's already there), initialize the file for writing, then store 0 in *outDuration*.

If *create* is false, you should open the existing file and put the actual running time of the file into *outDuration*.

Return **B_OK** if successful; otherwise, return a negative error code, as appropriate. If you forget to implement these functions, they'll always return **B_ERROR**.

SetRef() see GetRef()

SniffRef()

protected:

```
virtual status_t SniffRef(const entry_ref &file, char *outMimeType,
    float *outQuality) = 0
```

When the system (or an application program) finds an unknown file, it can call BMediaRoster to try to identify the file. Your node's `SniffRef()` function will be called so your node can look at the file.

Inspect the file referenced by *file*. If you can handle the format, set *outMimeType* to the MIME type of the file format (the buffer is 256 bytes long), and set *outQuality* to indicate how well you can process the file (where 0.0 means you can't handle it at all and 1.0 means the file format is a privately-owned format that your application handles perfectly).

If you know how to identify the file, but not how to play or record it, return 0.0 for *outQuality*.

If you know how to decode only some parts of the file, but not all of them, set outQuality to 0.4 or to 0.6 if you can handle both audio and video contained in the file, but not other forms of media data that are present.

Set *outQuality* to 0.9 if you can handle the format very well, but it's a publicly-defined format whose specifications you don't control. You should only return an *outQuality* value of 1.0 if the file format specifications are in your control and you implement them perfectly.

Interpolate between these values to give the best estimate you can of your ability to handle the file, so selection of the most suitable node can be made. For instance, if your node does low-quality but high-speed processing of an audio format, you might subtract a little bit from the *outQuality* value you might otherwise return.

Return `B_OK` if you successfully sniff the file and have something to say about its contents; otherwise return an appropriate error code. If you don't know the file format at all, you should return `B_MEDIA_NO_HANDLER`.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFileInterface Function Summary

Function Summary

Function	Description
DisposeFileFormatCookie()	Hook: dispose of the file format iterator cookie, if necessary
BFileInterface()	Constructor
GetDuration()	Hook: return the duration in microseconds of the media in the file being worked with
GetNextFileFormat()	Hook: return the next file format in an iteration loop through all formats you support
GetRef()	Hook: returns the entry_ref of the file currently being worked on
HandleMessage()	Handles messages received on the node's control port
SetRef()	Hook: specifies the file to be worked with.
SniffRef()	Hook: given a file, report its type and how well the node can handle it

---

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Storage Kit

BMimeType

Declared in: be/storage/Mime.h
more...

GetSnifferRule() , SetSnifferRule() , CheckSnifferRule()

```
status_t GetSnifferRule(BString *result) const
status_t SetSnifferRule(const char *rule)
static status_t CheckSnifferRule(const char *rule, BString *parseError)
```

`GetSnifferRule()` and `SetSnifferRule()` get and set the pattern-matching rule that the file type sniffer uses to detect the MIME type represented by this object. The sniffer *rule* is a string that follows this format:

```
rating [begin:end] ( [begin:end] 'pattern1' | [begin:end] 'pattern2' | ... )
```

- *rating* is a mandatory float between 0.0 and 1.0 that describes the priority of the rule, compared to other rules for the same type.
- *[begin:end]* is the optional "global byte range" declaration. The sniffer will look for a matching pattern that begins within this range of data. The *[begin:end]* that appears outside the parenthesis (i.e. just after *rating*) applies to all the patterns for this rule. If a pattern must begin at a particular byte offset, you can exclude the ":end" declaration.
- *'pattern1'*, *'pattern2'*, etc. These are the patterns themselves. You can OR patterns with the "|" operator. (You can't AND patterns.) An immediately preceding byte range declaration applies to the following pattern (only), and overrides the global byte range declaration.

If no byte range declaration is provided, [0:0] is assumed—i.e. the pattern(s) must start at the first byte.

For example, the following rule strings says that the content of an examined file must begin with the characters 'ABCD':

```
"1.0 ('ABCD')"
```

This rule says that the content must contain the characters 'ABCD' or 'abcd' starting somewhere in the first three bytes:

```
"1.0 [0:3] ('ABCD' | 'abcd')"
```

Here the content must contain 'ABCD' or 'abcd' starting in the first three bytes, or it must contain 'EFGH' starting at byte 13:

```
"1.0 [0:3] ('ABCD' | 'abcd' | [13] 'EFGH')"
```

The **CheckSnifferRule()** function is a debugging tool that checks your sniffer rule (independent of MIME type) and returns a string that describes the error. If the rule is correct, you get back (in *result*) a string that says "rule is correct". Otherwise, the result string contains an error message with a pointer to the offending part of the rule. The string is meant to be viewed in a monospace font.

B_SNIFFER_RULE_CHANGED

```
enum { ... B_SNIFFER_RULE_CHANGED, ... };
```

B_SNIFFER_RULE_CHANGED is a new MIME database-watching bit vector value. If your application has called **BMimeType::StartWatching()** and someone changes a sniffer rule, your target will receive a **B_META_MIME_CHANGED** message that has, as in its "be:which" field, the value **B_SNIFFER_RULE_CHANGED**.

GuessMimeType()

```
static status_t GuessMimeType(const entry_ref *file, BMimeType *result)
static status_t GuessMimeType(const void *buffer, int32 length, BMimeType *result)
static status_t GuessMimeType(const char *filename, BMimeType *result)
```

These new class functions ask the sniffer to identify the MIME type of a file or data. *result* must be allocated before it's passed in.

BNode

Declared in: be/storage/Node.h

WriteAttrString() , ReadAttrString()

```
status_t WriteAttrString(const char *attr, const BString *string)
status_t ReadAttrString(const char *attr, BString *result) const
```

New functions to more easily work with string-valued file attributes. `WriteAttrString()` writes a `NULL`-terminator at the end of the string.

BPath

Declared in: be/storage/Path.h

BPath has new versions of the constructor and the **SetTo()** function that accepts **entry_ref** arguments.

BPath()

```
BPath(const entry_ref *ref)
```

SetTo()

```
status_t SetTo(const entry_ref *ref)
```

BQuery

Declared in: be/storage/Query.h

GetPredicate()

```
status_t GetPredicate(BString *result)
```

This new version of `GetPredicate()` returns the query's predicate string as a BString object.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Message Formats

The Media Kit has a number of message formats. The following table lists the messages (by their 'what' constants) and describes their intents. The subsequent sections list the fields that are contained in the messages.

Notification Messages

Declared in: be/media/MediaDefs.h

Constant	Meaning
B_MEDIA_WILDCARD	Used to match any notification (used only in calls to StartWatching() a StopWatching()).
B_MEDIA_NODE_CREATED	A new node has been created
B_MEDIA_NODE_DELETED	Not sent in R4.5.
B_MEDIA_CONNECTION_MADE	Not sent in R4.5.
B_MEDIA_CONNECTION_BROKEN	Not sent in R4.5.
B_MEDIA_BUFFER_CREATED	Not sent in R4.5.
B_MEDIA_BUFFER_DELETED	Not sent in R4.5.
B_MEDIA_TRANSPORT_STATE	Not sent in R4.5.
B_MEDIA_PARAMETER_CHANGED	Indicates that a parameter in a BParameterWeb has changed.
B_MEDIA_FORMAT_CHANGED	Not sent in R4.5.
B_MEDIA_WEB_CHANGED	Indicates that a node's BParameterWeb has changed.
B_MEDIA_DEFAULT_CHANGED	Not sent in R4.5.
B_MEDIA_NEW_PARAMTER_VALUE	Indicates that a parameter in a BParameterWeb has changed. Unlike B_MEDIA_PARAMETER_CHANGED notification, this includes the new value.
B_MEDIA_NODE_STOPPED	Indicates that a node has stopped.
B_MEDIA_FLAVORS_CHANGED	Indicates that flavors have changed.

Messages with these 'what' are sent by the media server to any application that's registered to receive notifications by calling BMediaRoster::StartWatching().

B_MEDIA_NODE_CREATED , B_MEDIA_NODE_DELETED

Field	Type Code	Description
"be:media_node_id"	B_INT32_TYPE	The ID of the media_node that was created or deleted.

B_MEDIA_CONNECTION_MADE

The B_MEDIA_CONNECTION_MADE message has three data fields:

Field	Type Code	Description
"be:output"	B_RAW_TYPE	A media_output structure describing the connection's output.
"be:input"	B_RAW_TYPE	A media_input structure describing the connection's input.
"be:format"	B_RAW_TYPE	A media_format structure describing the format of the data that flows across the connection.

B_MEDIA_CONNECTION_BROKEN

Field	Type Code	Description
"destination"	B_RAW_TYPE	A media_destination structure describing the destination of the connection that's been broken.
"source"	B_RAW_TYPE	A media_source structure describing the source of the connection that's been broken.

B_MEDIA_BUFFER_CREATED

Field	Type Code	Description
"clone_info"	B_RAW_TYPE	An area_info structure describing the buffer's location in memory. This is an array, one entry per buffer created.

B_MEDIA_BUFFER_DELETED

Field	Type Code	Description
"be:media_buffer_id"	B_INT32_TYPE	The buffer ID number of the buffer being deleted. This is an array, one entry per buffer deleted.

B_MEDIA_FORMAT_CHANGED

Field	Type Code	Description
"be:source"	B_RAW_TYPE	A media_source structure describing the source of the connection whose format changed.
"be:destination"	B_RAW_TYPE	A media_source structure describing the source of the connection whose format changed.
"be:format"	B_RAW_TYPE	A media_format structure describing the new format.

B_MEDIA_PARAMETER_CHANGED

Field	Type Code	Description
"be:node"	B_RAW_TYPE	A media_node structure indicating which node's parameter web has changed.
"be:parameter"	B_INT32_TYPE	The ID number of the parameter whose value has changed.

B_MEDIA_WEB_CHANGED

Field	Type Code	Description
"node"	B_RAW_TYPE	A media_node structure indicating which node's parameter web has changed.

B_MEDIA_NEW_PARAMETER_VALUE

Field	Type Code	Description
"node"	B_RAW_TYPE	A media_node structure indicating which node is reporting a changed parameter value.
"parameter"	B_INT32_TYPE	The parameter ID of the changed parameter.
"when"	B_INT64_TYPE	The performance time at which the change took effect.
"value"	B_RAW_TYPE	The parameter's new value.

B_MEDIA_FLAVORS_CHANGED

Field	Type Code	Description
"be:addon_id"	B_INT32_TYPE	The add-on ID of the add-on whose flavors have changed
"be:new_count"	B_INT32_TYPE	How many new flavors have been added.
"be:gone_count"	B_INT32_TYPE	How many flavors have been removed.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAppFileInfo Function Summary

Function Summary

Function	Description
BAppFileInfo()	Constructor
~BAppFileInfo()	Destructor
GetAppFlags()	Returns application flags
GetIcon()	Returns application icons
GetIconForType()	Returns icon used by an application to display a given file type
GetPreferredApp()	Sets the preferred app for this executable
GetSignature()	Returns the application signature
GetSupportedTypes()	Returns the application's supported file types
GetType()	Returns the application's file type
GetVersionInfo()	Returns the application's version information
InitCheck()	Returns the status of the most recent initialization
IsSupportedType()	Tests whether an application supports a given MIME type
IsUsingAttributes()	Tests whether file information will be stored as attributes
IsUsingResources()	Tests whether file information will be stored as resources
SetAppFlags()	Sets application flags
SetIcon()	Sets application icons

SetIconForType()	Sets icon used by an application to display a given file type
SetInfoLocation()	Sets the location where the file information is to be stored
SetPreferredApp()	Gets the preferred app for this executable
SetSignature()	Sets the application signature
SetSupportedTypes()	Sets the application's supported file types
SetTo()	Initializes the object by pointing it to a BFile
SetType()	Sets the application's file type
SetVersionInfo()	Sets the application's version information
Supports()	Tests whether an application supports a given MIME type

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Variables, Constants, and Defined Types

This section lists the global variables, constants, and defined types that are defined in the Application Kit. Error codes are documented in the chapter on the Support Kit.

Although the Application Kit defines the constants for all system messages (such as `B_REFS_RECEIVED` and `B_KEY_DOWN`), only those that objects in this kit handle are listed here. Those that designate interface messages are documented in the chapter on the Interface Kit.

Global Variables

be_app

Declared in: be/app/Application.h

```
BApplication *be_app
```

This variable provides global access to the BApplication object. It's initialized by the BApplication constructor.

See also: the BApplication class

be_app_messenger

Declared in: be/app/Application.h

```
BMessenger *be_app_messenger
```

This variable provides global access to a BMessenger object whose target is **be_app**. It's initialized by the BApplication constructor.

See also: the BApplication class

be_clipboard

Declared in: be/app/Clipboard.h

```
BClipboard *be_clipboard
```

This variable gives applications access to the system clipboard—the shared repository of data for cut, copy, and paste operations. It's initialized at startup.

See also: the BClipboard class

be_roster

Declared in: be/app/Roster.h

```
const BRoster *be_roster
```

This variable points to the application's global BRoster object. The BRoster keeps a roster of all running applications and can add applications to the roster by launching them. It's initialized when the application starts up.

See also: the BRoster class

Constants

Application Flags

Declared in: be/app/Roster.h

Constant
B_BACKGROUND_APP
B_ARGV_ONLY
B_LAUNCH_MASK

These constants are used to get information from the `flags` field of an **`app_info`** structure.

See also: **`BRoster::GetAppInfo()`**, "Launch Constants" below

Application Messages

Declared in: be/app/AppDefs.h

Constant
B_QUIT_REQUESTED
B_READY_TO_RUN
B_APP_ACTIVATED
B_ABOUT_REQUESTED
B_QUIT_REQUESTED
B_ARGV_RECEIVED
B_REFS_RECEIVED
B_PULSE

These constants represent the system messages that are recognized and given special treatment by BApplication and BLooper dispatchers. Application messages concern the application as a whole, rather than any particular window thread. See the introduction to this chapter and the BApplication class for details.

See also: "Application Messages" on page 30 of the BApplication class

Cursor Constants

Declared in: be/app/AppDefs.h

const unsigned char **B_HAND_CURSOR**[]
const unsigned char **B_I_BEAM_CURSOR**[]

These constants contain all the data needed to set the cursor to the default hand image or to the standard I-beam image for text selection.

See also: `BApplication::SetCursor()`

---

filter_result Constants

Declared in: be/app/MessageFilter.h

Constant
B_SKIP_MESSAGE
B_DISPATCH_MESSAGE

These constants list the possible return values of a filter function.

See also: `BMessageFilter::Filter()`

Launch Constants

Declared in: be/app/Roster.h

Constant
B_MULTIPLE_LAUNCH
B_SINGLE_LAUNCH
B_EXCLUSIVE_LAUNCH

These constants explain whether an application can be launched any number of times, only once from a particular executable file, or only once for a particular application signature. This information is part of the `flags` field of an `app_info` structure and can be extracted using the `B_LAUNCH_MASK` constant.

See also: `BRoster::GetAppInfo()`, "Application Flags" above

Looper Port Capacity

Declared in: be/app/Looper.h

Constant
B_LOOPER_PORT_DEFAULT_CAPACITY

This constant records the default capacity of a BLooper's port. The default is 100 slots; a greater or smaller number can be specified when constructing the BLooper.

See also: the BLooper constructor

Message Constants

Declared in: be/app/AppDefs.h

Constant
B_REPLY
B_NO_REPLY
B_MESSAGE_NOT_UNDERSTOOD
B_SAVE_REQUESTED
B_CANCEL
B_SIMPLE_DATA
B_MIME_DATA
B_ARCHIVED_OBJECT
B_UPDATE_STATUS_BAR
B_RESET_STATUS_BAR
B_NODE_MONITOR
B_QUERY_UPDATE
B_CUT
B_COPY
B_PASTE
B_SELECT_ALL
B_SET_PROPERTY
B_GET_PROPERTY
B_CREATE_PROPERTY
B_DELETE_PROPERTY
B_GET_SUPPORTED_SUITES

These constants mark messages that the system sometimes puts together, but that aren't dispatched like system messages. See "Standard Messages" in the **Message Protocols** appendix for details.

See also: `BMessage::SendReply()`, the BTextView class in the Interface Kit

---

message_delivery Constants

Declared in: be/app/MessageFilter.h

Constant
B_ANY_DELIVERY
B_DROPPED_DELIVERY
B_PROGRAMMED_DELIVERY

These constants distinguish the delivery criterion for filtering a BMessage.

See also: the BMessageFilter constructor

message_source Constants

Declared in: be/app/MessageFilter.h

Constant
B_ANY_SOURCE
B_REMOTE_SOURCE
B_LOCAL_SOURCE

These constants list the possible constraints that a BMessageFilter might impose on the source of the messages it filters.

See also: the BMessageFilter constructor

Message Specifiers

Declared in: be/app/Message.h

Constant
B_NO_SPECIFIER
B_DIRECT_SPECIFIER
B_INDEX_SPECIFIER
B_REVERSE_INDEX_SPECIFIER
B_RANGE_SPECIFIER
B_REVERSE_RANGE_SPECIFIER
B_NAME_SPECIFIER
B_ID_SPECIFIER
B_SPECIFIERS_END = 128

These constants fill the `what` slot of specifier BMessages. Each constant indicates what other information the specifer contains and how it should be interpreted. For example, a **B_REVERSE_INDEX_SPECIFIER** message has an "index" field with an index that counts backwards from the end of a list. A **B_NAME_SPECIFIER** message includes a "name" field that names the requested item.

Defined Types

app_info

Declared in: be/app/Roster.h

```
typedef struct {
        thread_id thread;
        team_id team;
        port_id port;
        uint32 flags;
        entry_ref ref;
        char signature[B_MIME_TYPE_LENGTH];
        app_info(void);
        ~app_info(void);
    } app_info
```

This structure is used by BRoster's `GetAppInfo()`, `GetRunningAppInfo()`, and `GetActiveAppInfo()` functions to report information about an application. Its constructor ensures that its fields are initialized to invalid values. To get meaningful values for an actual application, you must pass the structure to one of the BRoster functions. See those functions for a description of the various fields.

See also: `BRoster::GetAppInfo()`

---

filter_result

Declared in: be/app/MessageFilter.h

```
typedef enum { . . . } filter_result
```

This type distinguishes between the `B_SKIP_MESSAGE` and `B_DISPATCH_MESSAGE` return values for a filter function.

See also: `BMessageFilter::Filter()`

---

message_delivery

Declared in: be/app/MessageFilter.h

```
typedef enum { . . . } message_delivery
```

This type enumerates the delivery criteria for filtering a message.

See also: the BMessageFilter constructor

---

message_source

Declared in: be/app/MessageFilter.h

```
typedef enum { . . . } message_source
```

This type enumerates the source criteria for filtering a message.

See also: the BMessageFilter constructor

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Keyboard Messages

B_KEY_DOWN , B_KEY_UP , B_UNMAPPED_KEY_DOWN , B_UNMAPPED_KEY_UP

Source: The system.
Target: The focus view's BWindow.
Hook: `BView::KeyDown()` (`B_KEY_DOWN`)
`BView::KeyUp()` (`B_KEY_UP`)
(The `...UNMAPPED...` messages don't map to hook functions.)

`B_KEY_DOWN` is sent when the user presses (or holds down) a key that's mapped to a character; `B_KEY_UP` is sent when the user releases the key. `B_UNMAPPED_KEY_DOWN` and `B_UNMAPPED_KEY_UP` are sent if the key isn't mapped to a character. This doesn't include modifier keys, which are reported in the `B_MODIFIERS_CHANGED` message.

Field	Type code	Description
"when"	`B_INT64_TYPE`	Event time, in microseconds since 01/01/70
"key"	`B_INT32_TYPE`	The code for the physical key that was pressed. See "More on Keyboard Mapping" for a discussion of the keymap.
"be:key_repeat" (`B_KEY_DOWN`	`B_INT32_TYPE`	The "iteration number" of this key down. When the

only)		user holds the key down, successive messages are sent with increasing key repeat values. This field isn't present in the initial event; the first repeat message (i.e., the second key down message) has a key repeat value of 1.
"modifiers"	B_INT32_TYPE	The modifier keys that were in effect at the time of the event. See "Modifier Keys" for a list of values.
"states"	B_UINT8_TYPE	The state of all keys at the time of the event. See "Key States."
"byte" [3] (B_KEY_DOWN and B_KEY_UP only)	B_INT8_TYPE	The UTF8 data that's generated
"bytes" (B_KEY_DOWN and B_KEY_UP only)	B_STRING_TYPE	The string that's generated. (The string usually contains a single character.)
"raw_char" (B_KEY_DOWN and B_KEY_UP only)	B_INT32_TYPE	Modifier-independent ASCII code for the character.

- B_KEY_UP see B_KEY_DOWN
- B_UNMAPPED_KEY_UP see B_KEY_DOWN

B_MODIFIERS_CHANGED

Source: The system.
Target: The focus view's window.

Sent when the user presses or releases a modifier key.

Field	Type code	Description
"when"	B_INT64_TYPE	Event time, in microseconds since 01/01/70
"modifiers"	B_INT32_TYPE	The current modifier keys. See "Modifier Keys" for a list of values.
"be:old_modifiers"	B_INT32_TYPE	The previous modifier keys.
"states"	B_UINT8_TYPE	The state of all keys at the time of the event. See "Key States.".

The Be Book Keyboard Keyboard Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Keyboard Table of Contents

- Keyboard Information
- Definitions
- Modifier Keys
- More on Keyboard Mapping
- Character Constants
- The Journey of a Keystroke
- Keyboard Messages
- Key Codes
- Key States

The Be Book Keyboard Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Be Book Keyboard

Keyboard: Master Index

C

B_CAPS_LOCK	Modifier Keys
Character Constants	anon
Character Constants	Character Constants
Character Constants	Character Constants
Character Keys	Definitions
B_COMMAND_KEY	Modifier Keys
B_CONTROL_KEY	Modifier Keys

D

Dead Keys	Definitions
Definitions	Definitions
Definitions	Definitions

J

The Journey of a Keystroke	anon
The Journey of a Keystroke	The Journey of a Keystroke
The Journey of a Keystroke	The Journey of a Keystroke

K

Key Codes	Key Codes
Key Codes	Key Codes
Key Codes	anon
Key Codes	Definitions
Key States	Key States
Key States	Key States
B_KEY_DOWN	Keyboard Messages
B_KEY_UP	Keyboard Messages
Keyboard Information	Keyboard Information
Keyboard Information	Keyboard Information
Keyboard Messages	Keyboard Messages
Keyboard Messages	Keyboard Messages
Keyboard Messages	anon
The Keymap	Definitions

L

B_LEFT_COMMAND_KEY	Modifier Keys
B_LEFT_CONTROL_KEY	Modifier Keys
B_LEFT_OPTION_KEY	Modifier Keys
B_LEFT_SHIFT_KEY	Modifier Keys

M

B_MENU_KEY	Modifier Keys
Modifier Keys	Definitions

Modifier Keys	Modifier Keys
Modifier Keys	Modifier Keys
B_MODIFIERS_CHANGED	Keyboard Messages
More on Keyboard Mapping	anon
More on Keyboard Mapping	More on Keyboard Mapping
More on Keyboard Mapping	More on Keyboard Mapping

N

B_NUM_LOCK	Modifier Keys

O

B_OPTION_KEY	Modifier Keys

R

Repeating Keys	Definitions
B_RIGHT_COMMAND_KEY	Modifier Keys
B_RIGHT_CONTROL_KEY	Modifier Keys
B_RIGHT_OPTION_KEY	Modifier Keys
B_RIGHT_SHIFT_KEY	Modifier Keys

S

The Scroll Lock Key	Definitions
B_SCROLL_LOCK	Modifier Keys

B_SHIFT_KEY	Modifier Keys
Step 1: The Input Server Add-on	The Journey of a Keystroke
Step 2: The Application Server	The Journey of a Keystroke
Step 3: The Active Window	The Journey of a Keystroke
Step 4: The Focused View	The Journey of a Keystroke

U

B_UNMAPPED_KEY_DOWN	Keyboard Messages
B_UNMAPPED_KEY_UP	Keyboard Messages

The Be Book Keyboard

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Modifier Keys

Conceptually, there are eight modifier keys. The global `modifiers()` function returns a value that encodes the current status of all eight of these keys. You can test for the state of a particular modifier key by comparing `modifiers()` with the constants listed in the table below. For example, here's how you can see if the user is holding down one of the **Shift** keys:

```
if (modifiers() | B_SHIFT_KEY) {
   /* a shift key is down */
}
```

The modifier mask is also included in the "modifiers" field in BMessages that report keyboard and mouse events.

How your application uses modifier keys is up to you. The table below lists the eight keys and how they're typically used.

Function	Description
B_SHIFT_KEY	Maps character keys to their uppercase or alternative versions.
B_COMMAND_KEY	In combination with a character key, performs a keyboard shortcut.
B_CONTROL_KEY	Turns alphabetic keys into control characters.
B_OPTION_KEY	Maps keys to alternative characters, typically characters in an extended set—multibyte UTF-8 characters.
B_MENU_KEY	Pops open a menu.
B_CAPS_LOCK	Reverses the effect of the **Shift** key on alphabetic characters.
B_NUM_LOCK	Reverses the effect of the **Shift** key on numeric characters.
B_SCROLL_LOCK	Stops the display from scrolling.

If you're interested in whether the left or right modifer key of a given type was pressed, you can check against these additional values:

B_LEFT_SHIFT_KEY	**B_RIGHT_SHIFT_KEY**
B_LEFT_CONTROL_KEY	**B_RIGHT_CONTROL_KEY**
B_LEFT_OPTION_KEY	**B_RIGHT_OPTION_KEY**
B_LEFT_COMMAND_KEY	**B_RIGHT_COMMAND_KEY**

Most of the modifiers map to single, specific keys: **Shift** and **Caps Lock** are examples. But others don't: the **Menu** key maps to **Shortcut**+**Escape** (where **Shortcut** is either **Alt**, **Command**, or **Control**, depending on the user's preferences and keyboard).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Character Constants

The Interface Kit defines constants for characters that aren't normally represented by a visible symbol. This includes the usual space and backspace characters, but most invisible characters are produced by the function keys and the navigation keys located between the main keyboard and the numeric keypad. The character values associated with these keys are more or less arbitrary, so you should always use the constant in your code rather than the actual character value. Many of these characters are also produced by alphabetic keys when a Control key is held down.

The table below lists character constants defined in the kit and the keys they're associated with.

Key label	Key code	Character reported
Backspace	0x1e	`B_BACKSPACE`
Tab	0x26	`B_TAB`
Enter	0x47	`B_ENTER`
space bar	0x5e	`B_SPACE`
Escape	0x01	`B_ESCAPE`
F1 – F12	0x02 through 0x0d	`B_FUNCTION_KEY`
Print Screen	0x0e	`B_FUNCTION_KEY`
Scroll Lock	0x0f	`B_FUNCTION_KEY`
Pause	0x10	`B_FUNCTION_KEY`
System Request	0x7e	0xc8
Break	0x7f	0xca
Insert	0x1f	`B_INSERT`
Home	0x20	`B_HOME`
Page Up	0x21	`B_PAGE_UP`
Delete	0x34	`B_DELETE`
End	0x35	`B_END`
Page Down	0x36	`B_PAGE_DOWN`
up arrow	0x57	`B_UP_ARROW`
left arrow	0x61	`B_LEFT_ARROW`
down arrow	0x62	`B_DOWN_ARROW`
right arrow	0x63	`B_RIGHT_ARROW`

Several keys are mapped to the `B_FUNCTION_KEY` character. An application can determine which function key was pressed to produce the character by testing the key code against these constants:

`B_F1_KEY`	`B_F6_KEY`	`B_F11_KEY`

B_F2_KEY	B_F7_KEY	B_F12_KEY
B_F3_KEY	B_F8_KEY	B_PRINT_KEY (the "Print Screen" key)
B_F4_KEY	B_F9_KEY	B_SCROLL_KEY (the "Scroll Lock" key)
B_F5_KEY	B_F10_KEY	B_PAUSE_KEY

Note that key 0x30 (*P*) is also mapped to B_FUNCTION_KEY when the **Control** key is held down.

Each of the character constants listed above is a one-byte value falling in the range of values where ASCII and Unicode overlap. For convenience, the Interface Kit also defines some constants for common characters that fall outside that range. These characters have multibyte representations in UTF-8, so the constant is defined as a character string. For example:

```
#define B_UTF8_OPEN_QUOTE"xE2x80x9C"
#define B_UTF8_CLOSE_QUOTE"xE2x80x9D"
#define B_UTF8_COPYRIGHT"xC2xA9"
```

See "Character Constants" in the "Global Variables, Constants, and Defined Types" section of **The Interface Kit** for a full list of these constants.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Definitions

This document lists and describes the keyboard terms and concepts that you should be familiar with.

Key Codes

Keyboards come in a variety of styles and sizes, with different numbers of keys on them, sometimes in different positions. Some special keys or symbols even appear more than once on the same keyboard—for example, most keyboards have two **Enter** keys (sometimes one or both of them is called **Return**). Most keyboards have two **Shift** keys, and many have two keys that can generate the "=" character—one of them on the numeric keypad.

Each of these keys is unique as far as the hardware is concerned, even if they usually generate the same result as far as the user is concerned. Because of this, each key has a unique number, called a **key code**. This is a numeric byte value between 0x01 and 0x7F. They're not the same as the ASCII or Unicode values for the characters generated by the keys—which makes sense, since not every key generates a character.

For example, the left **Shift** key has the key code value of 0x4B, and the right **Shift** key has the key code of 0x56. Neither of these generates a `B_KEY_UP` event (although they do generate a `B_MODIFIERS_CHANGED` event). The additional implication here is clear: software can actually tell the difference between these two keys—and any other keys that generate the same Unicode value—if they need to do so.

It's also important to note that key codes don't differentiate between shifted and unshifted keys. Pressing "A" generates

the key code 0x27 whether the **Caps Lock** or **Shift** key is down or not.

The Keymap

Most software written these days expects keys to be received in either ASCII or Unicode format (or, in the case of BeOS, in UTF-8 format, which is a form of Unicode in which the ASCII codes are contained). So obviously receiving a value of 0x27 for the "A" key isn't going to help a program that expects to receive 0x41, the UTF-8 value for the character "A".

So clearly the key codes the input server add-on obtains from the keyboard device need to be translated into UTF-8. This is done by using a **keymap**. This is a table that can be used to look up a key code and determine, based on which modifier keys are currently engaged, the UTF-8 bytes that should be generated. It also defines which keys are treated as modifier keys.

BeOS maintains a keymap for use by all applications. Applications that want to look up key codes in the keymap can call `get_key_map()` to get a copy of it. The **Keymap** preference panel can be used to configure the system keymap.

Character Keys

A character key is a key that's mapped to a particular UTF-8 byte or byte sequence. For example, in the standard American keymap, the key labeled "A" is mapped to the letter "a" when pressed with no modifier keys down, "A" when the **Caps Lock** or a **Shift** key is down, and control-A (UTF-8 code 0x01) when the **Control** key is down. Mapped character keys generate `B_KEY_DOWN` and `B_KEY_UP` messages when they're pressed and released.

Keys that aren't mapped to characters generate `B_UNMAPPED_KEY_DOWN` and `B_UNMAPPED_KEY_UP` messages.

Modifier Keys

Modifier keys set states that affect the way character keys are interpreted. Some modifier keys, such as **Caps Lock**, toggle in and out of a locked-on or locked-off position each time they're pressed. Others, like **Shift**, set the state only as long as they're held down.

The keymap contains lists of how to interpret each key depending on which modifier keys are currently engaged.

The Scroll Lock Key

The **Scroll Lock** key is unique. In the standard keyboard Input Server add-on, it both generates a character and sets a modifier state.

Repeating Keys

When a key is pressed and held down, it produces a continuous series of `B_KEY_DOWN` or `B_UNMAPPED_KEY_DOWN` messages, as long as the key is still held down and doesn't press another key. After the first message, there's a slight delay before the key begins repeating. This delay can be determined by calling `get_key_repeat_delay()` and can be changed by calling `set_key_repeat_delay()` (although you should leave the user's preference for this alone).

After the key begins to repeat, the `B_KEY_DOWN` or `B_UNMAPPED_KEY_DOWN` messages are sent at a fixed rate; this rate can be determined by calling `get_key_repeat_rate()`.

All keys repeat except **Pause**, **Break**, **Caps Lock**, **Num Lock**, and **Scroll Lock**.

Dead Keys

Dead keys don't generate characters until the user strikes another key. If the key the user presses after the dead key is in a particular set, the two keys together produce one key-down event (and, usually, one character). If the second keystroke isn't in that set, two key-down events are generated.

Dead keys are only dead when certain prescribed modifiers (by default, just the **Option** key) are held down. They're most appropriate when the character to be generated can be thought of as being composed of two distinguishable parts —such as "a" and "e" combining into "æ".

The system permits up to five dead keys. By default, they're reserved for combining diacritical marks with other characters. The diacritical marks are the acute (xab) and grave (') accents, dieresis (xac), circumflex (xab), and tilde (xf7).

The Be Book Keyboard Keyboard Index

Copyright © 2000 Be, Inc. All rights reserved..

The Journey of a Keystroke

When the user presses a key, a series of events is launched to handle that action. Let's follow the bits through the system after the user presses a key on the keyboard.

Step 1: The Input Server Add-on

When the user presses a key on the keyboard, the entity responsible for recognizing this action is the Input Server add-on for the keyboard device on which the key was pressed. The add-on uses the current keymap (which it has obtained by calling `get_key_map()`) to map the keycode to the character.

The add-on then constructs a `B_KEY_DOWN` or `B_UNMAPPED_KEY_DOWN` BMessage, containing both the raw keycode and the actual characters generated by the key, and sends the message to the Application Server by calling the `BInputServerDevice::EnqueueMessage()` function.

Step 2: The Application Server

The Aplication Server receives the message from the Input Server add-on and decides what to do with it. If the key pressed is something the Application Server traps, like a workspace change keystroke, it gets handled internally.

Otherwise, the Application server locates the active window and passes the BMessage along to its **`BWindow::DispatchMessage()`** function.

Step 3: The Active Window

The **`BWindow::DispatchMessage()`** function receives the BMessage from the Application Server, and intercepts some keys (like menu shortcuts and the Command-W close window sequence). If it's not a key the BWindow class (or derived class) wants to intercept, DispatchMessage passes the message along to the currently focused view's **`BView::MessageReceived()`** function.

Step 4: The Focused View

The message finally arrives at the view that's currently focused in the active window. This view's `MessageReceived()` function processes the keypress in whatever way it wants to.

Likewise, when the key is released, a `B_KEY_UP` or `B_UNMAPPED_KEY_UP` event is sent through this same route.

The Be Book Keyboard Keyboard Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerDevice

Derived from: *none*
Declared in: be/add-ons/input_server/InputServerDevice.h
Library: The Input Server
Allocation: By the Input Server only
Summary: *more...*

BInputServerDevice is a base class for *input devices*; these are instances of BInputServerDevice subclasses that generate *input events*. In most cases, an input device corresponds to a device driver that handles a specific brand or model of hardware (mouse, keyboard, tablet, etc.), but it doesn't have to: an input device can get events from the Net, or generate them algorithmically, for example. Also, a single BInputServerDevice can handle more than one device driver.

BInputServerDevice objects are created and deleted by the Input Server only—you never create or delete these objects themselves.

Starting and Sending Messages

For each device that your object registers, it gets a `Start()` function call. This is the Input Server's way of telling your object that it can begin generating input events (for the designated device). So far, all of this—from the add-on load to the `Start()` call—happens within a single Input Server thread (for all input devices). When your `Start()` function is called, you should spawn a thread so your object can generate events without blocking the Server. Events are generated and sent through the `EnqueueMessage()` function.

Device Types and Control Messages

The Input Server knows about two types of devices: keyboards, and pointing devices (mice, tablets, etc). When you register your object's devices (through `RegisterDevices()`) you have to indicate the *device type*. The Input Server uses the device type to predicate the *input device control messages* it sends to the devices. These messages, delivered in `Control()` calls, tell a device that there's been a change downstream that applies specifically to that type of device. For example, when the user changes the mouse speed, each pointing device receives a `B_MOUSE_SPEED_CHANGED` notification.

The Be-defined control messages are predicated on device type only.

If your BInputServerDevice object manages a device other than a pointer or a keyboard, you tell the Input Server that the device is undefined. In this case, the Input Server won't send your device any device-specific messages; to send your device a message you (or an application that knows about your device) have to use a BInputDevice object.

Pointing Devices

Pointing devices such as mice, trackballs, drawing tablets, etc. generate `B_MOUSE_MOVED` messages (which trigger a BView's `MouseMoved()` function) featuring a *where* field representing the cursor's location in view co-ordinates. Unfortunately, your BInputServerDevice doesn't know anything about views; that's the App Server's job. You'll still need to add this information to the `B_MOUSE_MOVED` messages generated by your BInputServerDevice, and the App Server will adjust it to view co-ordinates for you.

When generating a `B_MOUSE_MOVED` message, you add *x* and *y* fields in one of two ways:

- an offset relative to the cursor's previous position (`B_INT32_TYPE` values)
- an absolute position expressed in the range 0.0 to 1.0 (`B_FLOAT_TYPE` values)

Mice always use relative locations; tablets can use either (though they usually provide absolute values).

Relative Locations

All mice (and some drawing tablets) express the pointer location relative to its previous position. If your pointing device is operating in relative co-ordinate mode, you add *x* and *y* entries as `B_INT32_TYPE` values in device-defined units. The App Server interprets these units as pixels, so you may need to scale your output:

```
int32 xVal, yVal;
...
event->AddInt32( "x", xVal );
event->AddInt32( "y", yVal );
```

Absolute Locations

Drawing tablets or other pointing devices that provide absolute locations add the *x* and *y* entries as **B_FLOAT_TYPE**s:

```
float xVal, yVal;
...
event->AddFloat( "x", xVal );
event->AddFloat( "y", yVal );
```

These values must be in the range [0.0 to 1.0]. The app_server scales them to the screen's co-ordinate system so (0.0, 0.0) is the left-top, and (1.0, 1.0) is the right-bottom of the screen. This lets the pointing device work with any screen resolution, automatically.

Now where?

When the Application Server receives one of these **B_MOUSE_MOVED** messages, it converts the *x* and *y* values into absolute values in the target view's co-ordinate system, and then throws away the *x* and *y* entries in the message. Because of this, and the fact that some applications might want more accurate positional information from tablets, fill in the *be:tablet_x* and *be:tablet_y* fields as well:

```
float xVal, yVal;
...
event->AddFloat( "x", xVal );
event->AddFloat( "y", yVal );
event->AddFloat( "be:tablet_x", xVal );
event->AddFloat( "be:tablet_y", yVal );
```

Other Useful Information

Pressure information is stored in the *be:tablet_pressure* field, as a float in the range [0.0 to 1.0] (minimum pressure to maximum pressure):

```
float pressure;
...
event->AddFloat( "be:tablet_pressure", pressure );
```

If the tablet supports tilt information, store it in *be:tablet_tilt_x* and *be:tablet_tilt_y*, scaling the information to the range [0.0 to 1.0]. A tilt of (-1.0, -1.0) tilts to the left-top, (1.0, 1.0) tilts to the right-bottom, and (0.0, 0.0) is no tilt.

```
float tilt_x, tilt_y;
...
event->AddFloat( "be:tablet_tilt_x", tilt_x );
event->AddFloat( "be:tablet_tilt_y", tilt_y );
```

Tablets with pens that support an eraser store the eraser's state in the *be:tablet_eraser* field. A value of 1 means the pen is reversed (i.e. the eraser is on), and 0 means it should behave

normally.

```
int32 erase_mode;
...
event->AddInt32( "be:tablet_eraser", erase_mode );
```

Device State

The `Control()` protocol is designed to accommodate queries (in addition to commands). Currently, however, the Input Server maintains the keyboard and pointing device state and answers these queries itself; it doesn't forward any of the Be-defined query messages. For example, when an application asks for the current mouse speed setting (through `get_mouse_speed()`), the query gets no further than the Input Server itself—it doesn't get passed as a control message to a pointing device.

If you're designing a BInputServerDevice that manages a keyboard or pointing device, you must keep in mind that your device is not responsible for its "Be-defined" state. The elements of the state—mouse speed, key map, etc.—correspond to the control messages listed in "Input Device Control Messages".

Dynamic Devices

As hardware devices are attached and detached from the computer, you can add and remove items from your BInputServerDevice's list of registered devices (by calling **`RegisterDevice()`**/**`UnregisterDevice()`**). But your object has to first *notice* that a physical device has been added or removed. It does this by placing a node monitor on the device directory (**/dev**). As a convenience—and to help conserve resources—the BInputServerDevice class provides the **`Start`**/**`StopMonitoringDevices()`** functions which install and remove node monitors for you.

Creating and Registering

To create a new input device, you must:

- create a subclass of BInputServerDevice
- implement the **`instantiate_input_device()`** C function to create an instance of your BInputServerDevice subclass
- compile the class and the function as an add-on
- install the add-on in one of the *input device directories*

At boot time, the Input Server loads the add-ons it finds in the input device directories. For each add-on it loads, the Server invokes **`instantiate_input_device()`** to get a pointer to the add-on's BInputServerDevice object. After constructing the object, the Server calls **`InitCheck()`** to give the add-on a chance to bail out if the constructor failed. If the add-on wants to continue, it calls **`RegisterDevices()`** (from within **`InitCheck()`**) to tell the Server which physical or virtual devices it handles.

Installing an Input Device

The input server looks for input devices in the "**input_server/devices**" subdirectories of **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.

- You can install your input devices in the latter two directories—i.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add-ons that are supplied by the BeOS.

Hook Functions

- `Control()`
- `InitCheck()`
- `Start()`
- `Stop()`
- `SystemShuttingDown()`

Constructor and Destructor

BInputServerDevice()

```
BInputServerDevice(void)
```

Creates a new BInputServerDevice object. You can initialize your object—set initial values, spawn (but not necessarily resume; do that in `Start()`) threads, open drivers, etc.—either here or in the `InitCheck()` function, which is called immediately after the constructor.

~BInputServerDevice()

```
~BInputServerDevice()
```

Deletes the BInputServerDevice object. The destructor is invoked by the Input Server only—you never delete a BInputServerDevice object from your own code. When the destructor is called, the object's devices will already be unregistered and `Stop()` will already have been called. If this object spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

Control()

```
virtual status_t Control(const char *name,
    void *cookie,
    uint32 command,
    BMessage *message)
```

The `Control()` hook function is invoked by the Input Server to send an *input device control message* or a Node Monitor message to this object. *name* and *cookie* are the readable name and pointer-to-whatever-you-want that you used when registering the device (with the `RegisterDevices()` function).

The function's return value is ignored.

Input Device Control Messages

An input device control message is sent when a downstream change needs to be propagated to an input device. For example, when the user resets the mouse speed (through the **Mouse** preference), a `B_MOUSE_SPEED_CHANGED` control message is sent to all objects that have registered a `B_POINTING_DEVICE` device (see `RegisterDevices()`). *name* and *cookie* identify the device that this message applies to. The control message itself is represented by the *command* constant, optionally supplemented by *message*.

See "Input Device Control Messages" for a list of the control messages that the BeOS defines, and instructions for how to respond to them. An application can send a custom control message through a BInputDevice object; see `BInputDevice::Control()` for details.

Node Monitor Messages

A Node Monitor message is sent if an entry is added to or removed from one of the device directories that the object is monitoring, as set through `StartMonitoringDevice()`. In this case, *name* and *cookie* are `NULL`, command is `B_NODE_MONITOR`, and *message* describes the file that was added or deleted. The message's **opcode** field will be `B_ENTRY_CREATED` or `B_ENTRY_REMOVED` (or, potentially but nonsensically, `B_ENTRY_MOVED`). For instructions on how to read these messages, see "The Node Monitor" in the Storage Kit (or click on the opcode constants).

EnqueueMessage()

```
status_t EnqueueMessage(BMessage *message)
```

Sends an event message to the Input Server, which passes it through the input methods and input filters before sending it to the App Server. The message you create should be

appropriate for the action you're trying to depict. For example, if the user presses a key, you should create and send a **B_KEY_DOWN** message. A list of the system-defined event messages that an input device is expected to create and send is given in "Input Device Event Messages".

RETURN CODES

- B_OK. The message was sent.
- *Anything else*. The connection to the App Server has been broken—this isn't good, and you may want to check the **is_computer_on_fire()** function found in the Kernel Kit.

InitCheck()

```
virtual status_t InitCheck(void)
```

Invoked by the Input Server immediately after the object is constructed to test the validity of the initialization. If the object is properly initialized (i.e. all required resources are located or allocated), this function should return **B_OK**. **Start()** will be invoked soon if you need to do any extra initialization. If the object returns non-**B_OK**, the object is deleted and the add-on is unloaded.

The default implementation returns **B_OK**.

RegisterDevices() , UnregisterDevices()

```
status_t RegisterDevices(input_device_ref **devices)
status_t UnregisterDevices(input_device_ref **devices)
```

RegisterDevices() tells the Input Server that this object is responsible for the listed *devices*. This means that when a control message is sent back upstream, the message—which is tagged as being relevant for a specific device, or type of device—will be forwarded (through the **Control()** hook) to the responsible BInputServerDevice object(s). Typically, you initially register your devices as part of the constructor or **InitCheck()**. Registration is cumulative—each **RegisterDevices()** call adds to the object's current list of devices.

UnregisterDevices() tells the Input Server that this object is no longer responsible for the listed devices. The devices are automatically unregistered when your object is deleted.

RegisterDevices() invokes **Start()** for each device in the *devices* list; **UnregisterDevices()** invokes **Stop()**.

For both functions, the *devices* list must be **NULL**-terminated, and the caller retains ownership of the list and its contents.

Note that the BeOS currently only targets the device types when sending a **Control()** message. For example, let's say you've registered two pointing devices and a keyboard:

```
status_t MyISDevice::InitCheck()
{
...
   input_device_ref **devices =
      (input_device_ref **)malloc(sizeof(*input_device_ref * 4));
   input_device_ref mouse1 = {"Mouse 1", B_POINTING_DEVICE,
                              (void *)this)};
   input_device_ref mouse2 = {"Mouse 2", B_POINTING_DEVICE,
                              (void *)this)};
   input_device_ref keyboard = {"Keyboard", B_KEYBOARD_DEVICE,
                              (void *)this)};
   devices[0] = &mouse1;
   devices[1] = &mouse2;
   devices[2] = &keyboard;
   devices[3] = NULL;
   RegisterDevices(devices);
   ...
}
```

When the user fiddles with the **Mouse** preference (more specifically, if an application calls **set_mouse_speed()** et. al.), this object will receive two **Control()** messages: one targets "Mouse 1", and the other targets "Mouse 2". That's because the mouse and keyboard functions (as defined by the BeOS and as used by the system preferences) know which type of device to control, but they don't provide a means for more granular identification. If you need a UI that identifies specific devices, you have to create the UI yourself, and use a BInputDevice object to tune the control messages that are sent back upstream.

RETURN CODES

- B_OK. At least one of the devices was registered.
- **B_ERROR**. None of the devices were registered.

The functions don't let you un/register the same device definition twice, and **RegisterDevices()** won't register a device that doesn't have a name (although the name can be ""). However, the functions don't complain about violations of these conditions as long as at least one definition is properly formed.

Start() , Stop()

virtual status_t **Start(**const char **name*, void **cookie***)*
virtual status_t **Stop(**const char **name*, void **cookie***)*

Start() is invoked by the Input Server to tell the object that it can begin sending events for the registered device identified by the arguments. The values of the arguments are taken from the **input_device_ref** structure you used to register the device (see **RegisterDevices()**). If your

object needs to resume a thread (spawned in the constructor, in `InitCheck()`, or here), this is the place to do it.

`Stop()` is invoked to tell the object to stop sending events for the registered device. The device is not unregistered—you can still receive `Control()` messages for the device while it's stopped. You should pause or kill any threads associated with the device (that were spawned by this object) from here.

The return value (for both of these functions) is ignored.

StartMonitoringDevice() , StopMonitoringDevice()

```
status_t StartMonitoringDevice(const char *deviceDir)
status_t StopMonitoringDevice(const char *deviceDir)
```

These are convenient covers for the Node Monitor's `watch_node()` and `stop_watching()` functions. You use them to watch for physical devices that are attached and detached, as indicated by changes to subdirectories of the system device directory (**/dev**).

deviceDir is the name of the device subdirectory that you want to watch. The "**/dev/**" root is automatically prepended; for example, if you want to watch for new ps2 mice, you would pass "**input/mouse/ps2**" as the *deviceDir* name. The Node Monitor is told to look for changes to the directory (`B_WATCH_DIRECTORY` opcode). When an entry is added or removed, this object receives a `B_NODE_MONITOR` message delivered to its `Control()` function.

RETURN CODES

- B_OK. Success.
- `B_ERROR`. Unspecified failure.
- `B_NOT_A_DIRECTORY`. You're trying to monitor a node that isn't a directory.
- `B_BAD_VALUE`. *deviceDir* not found.

SystemShuttingDown()

```
virtual status_t SystemShuttingDown(void) const
```

Tells the object that the Input Server is in the process of shutting down. Unless something interrupts the shutdown, this notification will be followed by a `Stop()` and **delete**, thus you don't have to do much from this function (other than note that the end is near).

The return value is ignored.

UnregisterDevices see RegisterDevices()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerDevice Function Summary

Function Summary

Function	Description
Control()	General notification hook from the Input Server
EnqueueMessage()	Sends an event message to the Input Server
InitCheck()	Tests whether the object was properly initialized
BInputServerDevice()	Constructor
~BInputServerDevice()	Destructor
RegisterDevices()	Tells the Input Server that this object corresponds to a set of devices
Start()	Tells the device to start generating events
StartMonitoringDevice()	Start monitoring a device directory
Stop()	Tells the device to stop generating events
StopMonitoringDevice()	Stop monitoring a device directory
SystemShuttingDown()	Tells the device that the Server is shutting up and shutting down
UnregisterDevices()	Tells the Input Server that this object no longer corresponds to a set of devices

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Input Server Messages

This section describes the messages that your Input Server objects are expected to create and send, and that they're expected to respond to.

Input Device Event Messages

Declared in: be/app/AppDefs.h
more...

This section lists the event messages that a BInputServerDevice is expected to create and send through its **`EnqueueMessage()`** function. The primary documentation for these messages is in the System Messages appendix (click on an item in the lists below to be taken to a specific definition).

Pointing Device Event Messages

`B_MOUSE_DOWN`
`B_MOUSE_UP`
`B_MOUSE_MOVED`

Note that a pointing device isn't expected to send the **`B_MOUSE_ENTER_EXIT`** message.

Keyboard Device Event Messages

`B_KEY_DOWN`
`B_UNMAPPED_KEY_DOWN`
`B_KEY_UP`
`B_UMAPPED_KEY_UP`
`B_MODIFIERS_CHANGED`

Input Device Control Messages

Declared in: be/add-ons/input_server/InputServerDevice.h

This section lists the control messages that are defined by the BeOS for pointing and keyboard devices. These are messages that appear in the **`BInputServerDevice::Control()`** function. Each control message is identified by the value that appears as the *command* argument in the **`Control()`** function. None of the Be-defined control messages use the additional BMessage argument.

Control messages are used to notify input devices of downstream requests. For example, when the user changes the mouse speed, a **`B_MOUSE_SPEED_CHANGED`** command is sent back upstream. It's expected that an input device that receives this message will tune subsequent event messages that it generates to match the requested mouse speed.

The messages listed below are defined by the BeOS; you can send custom control messages back upstream through the **`BInput::Control()`** function. Of course, this is only effective if you install a custom input device that can handle the messages.

Note that the Be-defined control messages ask a device to set parameters (such as mouse speed), but they never ask a device for the value of a parameter. For example, a pointing device is never asked what the mouse speed is. This is because the Input Server maintains the state of the

keyboard and pointing device environments and can answer these requests itself.

Furthermore, the Be-defined control messages don't contain the value of the parameter that's being set. For example, the `B_MOUSE_SPEED_CHANGED` message doesn't contain the requested mouse speed. The input device must ask the Input Server for the new value through a global function (`get_mouse_speed()`, in this case). The functions that correspond to the messages are listed in the descriptions below.

Pointing Device Control Messages

B_CLICK_SPEED_CHANGED

Requests that the receiver change the mouse double-click speed to the value retrieved through `get_click_speed()`.

B_MOUSE_MAP_CHANGED

Requests that the receiver change the mouse map (the correspondence between physical mouse buttons and the `B_PRIMARY_MOUSE_BUTTON`, et. al., constants) to the map retrieved through `get_mouse_map()`.

B_MOUSE_SPEED_CHANGED

Requests that the receiver change the mouse speed to the value retrieved through `get_mouse_speed()`.

B_MOUSE_TYPE_CHANGED

Requests that the receiver change the mouse type (the number of buttons) to the type retrieved through `get_mouse_type()`.

Keyboard Device Control Messages

B_KEY_LOCKS_CHANGED

Requests that the receiver change the state of the locked keys (caps lock, num lock, etc.). To get the desired state of the locking keys, read the states out of the key map returned by `get_key_map()`.

B_KEY_MAP_CHANGED

Requests that the receiver change the keyboard's key map—the mapping between physical keys and the character codes they generate. The new key map is returned by `get_key_map()`.

B_KEY_REPEAT_DELAY_CHANGED

Requests that the receiver change the delay before a held key starts generating repeated characters to the value retrieved through `get_key_repeat_delay()`.

B_KEY_REPEAT_RATE_CHANGED

Requests that the receiver change the speed at which a held key generates repeated characters to the value retrieved through `get_key_repeat_rate()`.

Device Monitoring

The `watch_input_devices()` function lets you ask the Input Server to send you a message when a device starts or stops, or when the set of registered devices changes. These "device monitoring" notifications are sent to the target specified in the function. The command constant is always `B_INPUT_DEVICES_CHANGED`. The *be:opcode* field will be one of:

B_INPUT_DEVICE_ADDED

An input device has been added to the system.

B_INPUT_DEVICE_REMOVED

An input device has been removed from the system.

B_INPUT_DEVICE_STARTED

An input device has been started.

B_INPUT_DEVICE_STOPPED

An input device has been stopped.

Input Method Events

Active input methods send input method events (`B_INPUT_METHOD_EVENT` messages) downstream to application views to help integrate the method's work with the view's display. Inside each `B_INPUT_METHOD_EVENT` message is a *be:opcode* field indicating the type of input method event:

B_INPUT_METHOD_CHANGED

Sent whenever the user changes the text during an input transaction.

B_INPUT_METHOD_LOCATION_REQUEST

Sent whenever the input method needs to know the on-screen locations of characters in the input transaction.

B_INPUT_METHOD_STARTED

Sent when a new input transaction is beginning.

B_INPUT_METHOD_STOPPED

Sent when an input transaction is completed.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerMethod

Derived from: BInputServerFilter
Declared in: be/add-ons/input_server/InputServerMethod.h
Library: libbe.so
Allocation: By the Input Server only
more...

BInputServerMethod is a base class for *input methods*; these are instances of BInputServerMethod that act as an interface between the user and languages using character sets that can't be easily represented on standard keyboards, such as the Japanese input method that comes with BeOS.

Input methods generally handle `B_KEY_DOWN` messages in their `Filter()` function (see BInputServerFilter), keeping some sort of state around to translate these standard keyboard messages into new `B_KEY_DOWN` messages representing another character set. An input method can handle any input event, they're not limited to keyboard events.

Writing an input method is an involved process, even though the BInputMethod protocol is relatively simple. If you're working on an input method, please feel free to contact Be Developer Technical Support (devsupport@be.com) for additional information.

Input Method Events

Input methods insert `B_INPUT_METHOD_EVENT` messages (using their `EnqueueMessage()` function) into the Input Server's event stream. These messages let BView subclasses work together with your input method to create a seamless experience for the user.

Each `B_INPUT_METHOD_EVENT` message contains a *be:opcode* field (an int32 value) indicating the kind of event:

- `B_INPUT_METHOD_STARTED`
- `B_INPUT_METHOD_STOPPED`
- `B_INPUT_METHOD_CHANGED`
- `B_INPUT_METHOD_LOCATION_REQUEST`

`B_INPUT_METHOD_STARTED` indicates that a new input transaction has begun. Add a BMessenger in the *be:reply_to* field; the receiver of the message will use this messenger to communicate with you during the transaction.

`B_INPUT_METHOD_STOPPED` indicates that the transaction is over.

In between the B_INPUT_METHOD_STARTED and B_INPUT_METHOD_STOPPED messages, you'll send B_INPUT_METHOD_CHANGED and B_INPUT_METHOD_LOCATION_REQUEST messages as the transaction proceeds.

B_INPUT_METHOD_CHANGED does most of the work in an input transaction; add the following important fields:

Entry	Type	Description
be:string	B_STRING_TYPE	The text the user is currently entering; the receiver will display it at the current insertion point. BTextView also highlights the text in blue to show that it's part of a transitory transaction.
be:selection	B_INT32_TYPE	A pair of B_INT32_TYPE offsets into the *be:string* if part of *be:string* is current selected. BTextView highlights this selection in red instead of drawing it in blue.
be:clause_start	B_INT32_TYPE	Zero or more offsets into the *be:string* for handling languages (such as Japanese) that separate a sentence or phrase into numerous clauses. An equal number of *be:clause_start* and *be:clause_end* pairs delimit these clauses; BTextView separates the blue/red highlighting wherever there is a clause boundary.
be:clause_end	B_INT32_TYPE	Zero or more offsets into *be:string*; there must be as many *be:clause_end* entries as there are *be:clause_start*.
be:confirmed	B_BOOL_TYPE	True when the user has entered and "confirmed" the current string and wishes to end the transaction. BTextView unhighlights the blue/red text and waits for a B_INPUT_METHOD_STOPPED (to close the transaction) or another B_INPUT_METHOD_CHANGED (to start a new transaction immediately).

B_INPUT_METHOD_LOCATION_REQUEST is the input method's way of asking for the on-screen location of each character in *be:string*. This information can be used by the input method to pop up additional windows giving the user an opportunity to select characters from a list or anything else that makes sense. When you send a B_INPUT_METHOD_LOCATION_REQUEST, the receiver will reply to the *be:reply_to* messenger (that you sent in your B_INPUT_METHOD_STARTED message) with a B_INPUT_METHOD_EVENT message, filling in the following fields:

Entry	Type	Description
be:opcode	B_INT32_TYPE	Set to B_INPUT_METHOD_LOCATION_REQUEST.
be:location_reply	B_POINT_TYPE	The co-ordinates of each character (there should be one *be:location_reply* for every character in *be:string*) relative to the display (not your view or your window).
be:height_reply	B_FLOAT_TYPE	The height of each character in *be:string*.

Creating

To create a new input method, you must:

- create a subclass of BInputServerMethod
- implement the **`instantiate_input_method()`** C function to create an instance of your BInputServerMethod subclass
- compile the class and function as an add-on
- install the add-on in one of the input method directories

At boot time (or whenever the Input Server is restarted; see "Dynamic Loading"), the Input Server loads the add-ons it finds in the input method directories. For each add-on it finds, the Server invokes **`instantiate_input_method()`** to get a pointer to the add-on's BInputServerMethod object. After constructing the object, the Server calls **`InitCheck()`** to give the add-on a chance to bail out if the constructor failed.

Installing an Input Method

The input server looks for input methods in the "**input_server/methods**" subdirectories of **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.

- You can install your input devices in the latter two directories—i.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add-ons that are supplied by the BeOS.

Hook Functions

- `MethodActivated()`

Constructor and Destructor

BInputServerMethod()

```
BInputServerMethod(const char *name, const uchar *icon)
```

Creates a new BInputServerMethod object. You can initialize the object—set initial values, spawn threads, etc.—either here or in the `InitCheck()` function, which is called immediately after the constructor.

name is a textual name describing the input method, and *icon* is the raw data for a 16x16 8-bit icon built from the standard BeOS palette. This *name* and *icon* will be displayed in the input method menu (the little keyboard icon in the Deskbar). When the user selects your input method from the menu, your `MethodActivated()` function is called.

~BInputServerMethod()

```
~BInputServerMethod()
```

Deletes the BInputServerMethod object. The destructor is invoked by the Input Server only—you never delete a BInputServerMethod object from your own code. If this object has spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

EnqueueMessage()

```
status_t EnqueueMessage(BMessage *message)
```

Inserts the specified *message* into the Input Server's event queue; the message continues down-stream from this point, passing through additional active input methods and input filters on its way to the App Server.

MethodActivated()

```
virtual status_t MethodActivated(bool active)
```

The MethodActivated() hook function is invoked by the Input Server when the user activates your input method (*active* is true) or deactivates it (*active* is false). This is your chance to activate any helper threads or loopers to the fact that you'll be handling input events soon.

Return **B_OK** if that's OK, or something else if it's not. The default implementation returns **B_OK**.

SetName()

```
status_t SetName(const char *name)
```

Changes your input method's name, as found in the input method menu of the Deskbar.

The Input Server makes a copy of *name*.

SetIcon()

```
status_t SetIcon(const uchar *icon)
```

Changes your input method's icon, as found in the input method menu of the Deskbar. icon should be raw data for a 16x16 8-bit icon built from the standard BeOS palette.

The Input Server makes a copy of *icon*.

SetMenu()

status_t SetMenu(const BMenu *_menu_, const BMessenger _target_)

Lets you assign a *menu* to your input method's entry in the input method menu of the Deskbar. Messages generated by the menu are sent to *target*.

Passing a **NULL** menu disables your input method's menu.

You retain ownership of *menu*; be careful not to delete it while the Input Server is still using it.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerFilter

Derived from: none
Declared in: be/add-ons/input_server/InputServerFilter.h
Library: libbe.so
Allocation: By the Input Server only
Summary: *more...*

BInputServerFilter is a base class for *input filters*; these are instances of BInputServerFilter that modify, generate, or eat *input events*. An input filter add-on is privy to all the events that pass through the Input Server's event stream. A filter is similar to the Interface Kit's BMessageFilter, but at a much lower level. The BInputServerFilter also sees *all* events, while a BMessageFilter only sees the events targeted at its BLooper. BInputServerFilters can also generate additional events in place of, or in addition to, the original input event.

BInputServerFilter objects are created and deleted by the Input Server only—you never create or delete these objects in your code.

Creating

To create a new input filter, you must:

- create a subclass of BInputServerFilter
- implement the **`instantiate_input_filter()`** C function to create an instance of your BInputServerFilter subclass
- compile the class and function as an add-on
- install the add-on in one of the input filter directories

At boot time (or whenever the Input Server is restarted; see "Loading" in **The Input Server**), the Input Server loads the add-ons it finds in the input filter directories. For each add-on it finds, the Server invokes **`instantiate_input_filter()`** to get a pointer to the add-ons's BInputServerFilter object. After constructing the object, the Server calls **`InitCheck()`** to give the add-on a chance to bail out if the constructor failed.

Installing an Input Filter

The input server looks for input filters in the "**input_server/filters**" subdirectories of **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.

- You can install your input devices in the latter two directories—i.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add-ons that are supplied with BeOS.

Hook Functions

- `Filter()`
- `InitCheck()`

Constructor and Destructor

BInputServerFilter()

```
BInputServerFilter(void)
```

Creates a new BInputServerFilter object. You can initialize the object—set initial values, spawn threads, etc.—either here or in the `InitCheck()` function, which is called immediately after the constructor.

~BInputServerFilter()

```
~BInputServerFilter()
```

Deletes the BInputServerFilter object. The destructor is invoked by the Input Server only—you never delete a BInputServerFilter object from your own code. If this object has spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

Filter()

virtual **filter_result Filter(**BMessage **message*, BList **outList***)**

The **Filter()** hook function is invoked by the Input Server to filter the in-coming *message*. You can discard the message, allow it to pass downstream (as is or modified), or replace it with one or more other messages:

- To discard *message*, implement **Filter()** to return **B_SKIP_MESSAGE** (don't delete the BMessage yourself—it's owned by the Input Server).
- If you implement **Filter()** to return **B_DISPATCH_MESSAGE** (and if you leave *outList* unchanged), *message* is allowed to continue downstream. You're allowed to modify (but not destroy or replace) *message*.
- To replace *message*, place one or more BMessages in *outList* and return **B_DISPATCH_MESSAGE**. *message* is discarded; the messages in *outList* are passed downstream in the order that they appear in the list. The Input Server owns all of the messages you pass back in *outList*, so don't delete them yourself. Note that if you want to include *message* in *outList*, you must copy the object before adding it to the list.

The default implementation returns **B_DISPATCH_MESSAGE** without modifying the *message*.

GetScreenRegion()

status_t **GetScreenRegion(**BRegion **region***)** const

The **GetScreenRegion()** function returns the screen's region in *region*. This is the most efficient way for an input filter to get the screen's region. The system screen saver's input filter uses this for its "sleep now"/"sleep never" corners.

GetScreenRegion() returns **B_OK**.

InputCheck()

virtual status_t **InitCheck(**void**)**

Invoked by the Input Server immediately after the object is constructed to test the validity of the initialization. If the object is properly initialized (i.e. all required resources are located or allocated), this function should return **B_OK**. If the object returns non-**B_OK**, the object is deleted and the add-on is unloaded.

The default implementation returns **B_OK**.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerFilter Function Summary

Function Summary

Function	Description
Filter()	Event filtering hook.
GetScreenRegion()	Get the screen region.
InputCheck()	Tests whether the object was properly initialized
~BInputServerFilter()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BList

Derived from: none
Declared in: be/support/List.h
Library: libbe.so
Summary: *more...*

A BList object is a compact, ordered list of data pointers. BList objects can contain pointers to any type of data, including—and especially—objects.

An item assigned to a BList is identified by an index to its position in the list. Indices start at 0 and are neither arbitrary nor permanent. If, for example, you insert an item into the middle of a list, the indices of the items at the tail of the list are incremented (by one). Similarly, removing an item decrements the indices of the following items.

A BList stores its items as type **`void *`**, so it's necessary to cast an item to the correct type when you retrieve it. For example, items retrieved from a list of BBitmap objects must be cast as BBitmap pointers:

```
BBitmap *theImage = (BBitmap *)myList->ItemAt(anIndex);
```

There's nothing to prevent you from adding a `NULL` pointer to a BList. However, functions that retrieve items from the list (such as **`ItemAt()`**) return `NULL` when the requested item can't be found. Thus, you can't distinguish between a valid `NULL` item and an invalid attempt to access an item that isn't there.

Constructor and Destructor

BList()

```
BList(int32 count = 20)
BList(const BList& anotherList)
```

Initializes the BList by allocating enough memory to hold *count* items. As the list grows and shrinks, additional memory is allocated and freed in blocks of the same size.

The copy constructor creates an independent list of data pointers, but it doesn't copy the pointed-to data. For example:

```
BList *newList = new BList(oldList);
```

Here, the contents of *oldList* and *newList*—the actual data pointers—are separate and independent. Adding, removing, or reordering items in *oldList* won't affect the number or order of items in *newList*. But if you modify the data that an item in *oldList* points to, the modification will be seen through the analogous item in *newList*.

The block size of a BList that's created through the copy constructor is the same as that of the original BList.

~BList()

```
virtual ~BList()
```

Frees the list of data pointers, but doesn't free the data that they point to. To destroy the data, you need to free each item individually:

```
void *anItem;
for ( int32 i = 0; anItem = myList->ItemAt(i); i++ )
   delete anItem;
delete myList;
```

See also: `MakeEmpty()`

Member Functions

AddItem()

bool **AddItem(**void **item*, int32 *index***)**
bool **AddItem(**void **item***)**

Adds an item to the BList at *index*—or, if no index is supplied, at the end of the list. If necessary, additional memory is allocated to accommodate the new item.

Adding an item never removes an item already in the list. If the item is added at an index that's already occupied, items currently in the list are bumped down one slot to make room.

If *index* is out of range (greater than the current item count, or less than zero), the function fails and returns **false**. Otherwise it returns **true**.

AddList()

bool **AddList(**BList **list*, int32 *index***)**
bool **AddList(**BList **list***)**

Adds the contents of another BList to this BList. The items from the other BList are inserted at *index*—or, if no index is given, they're appended to the end of the list. If the index is out of range, the function fails and returns **false**. If successful, it returns **true**.

See also: AddItem()

CountItems()

int32 **CountItems(**void**)** const

Returns the number of items currently in the list.

DoForEach()

void **DoForEach(**bool (**func*)(void **)**)**
void **DoForEach(**bool (**func*)(void **, void **), void **arg2***)**

Calls the *func* function once for each item in the BList. Items are visited in order, beginning with the first one in the list (index 0) and ending with the last. If a call to *func* returns **true**, the iteration is stopped, even if some items have not yet been visited.

func must be a function that takes one or two arguments. The first argument is the currently-considered item from the list; the second argument, if *func* requires one, is passed to `DoForEach()` as *arg2*.

FirstItem()

```
void *FirstItem(void) const
```

Returns the first item in the list, or `NULL` if the list is empty. This function doesn't remove the item from the list.

See also: `LastItem()`, `ItemAt()`

HasItem()

```
bool HasItem(void *item) const
```

Returns **`true`** if *item* is in the list, and **`false`** if not.

IndexOf()

```
int32 IndexOf(void *item) const
```

Returns the index where a particular *item* is located in the list, or a negative number if the *item* isn't in the list. If the item is in the list more than once, the index returned will be the position of its first occurrence.

IsEmpty()

```
bool IsEmpty(void) const
```

Returns **`true`** if the list is empty (if it contains no items), and **`false`** otherwise.

See also: `MakeEmpty()`

ItemAt()

```
void *ItemAt(int32 index) const
```

Returns the item at *index*, or `NULL` if the index is out of range. This function doesn't remove the item from the list.

See also: `Items()`, `FirstItem()`, `LastItem()`

Items()

```
void *Items(void) const
```

Returns a pointer to the BList's list. You can index directly into the list if you're certain that the index is in range:

```
myType *item = (myType *)Items()[index];
```

LastItem()

```
void *LastItem(void) const
```

Returns the last item in the list without removing it. If the list is empty, this function returns `NULL`.

See also: `RemoveLastItem()`, `FirstItem()`

MakeEmpty()

```
void MakeEmpty(void)
```

Empties the BList of all its items, without freeing the data that they point to.

See also: `IsEmpty()`, `RemoveItem()`

RemoveItem() , RemoveItems()

```
bool RemoveItem(void *item)
void *RemoveItem(int32 index)
```

```
bool RemoveItems(int32 index, int32 count)
```

`RemoveItem()` removes an item from the list. If passed an *item*, the function looks for the item in the list, removes it, and returns **`true`**. If it can't find the item, it returns **`false`**. If the item is in the list more than once, this function removes only its first occurrence.

If passed an *index*, `RemoveItem()` removes the item at that index and returns it. If there's no item at the index, it returns `NULL`.

`RemoveItems()` removes a group of *count* items from the list, beginning with the item at *index*. If the index is out of range, it fails and returns `false`. Otherwise, it removes the items, without checking to be sure that the list actually holds that many items at the index, and returns `true`.

The list is compacted after an item is removed. Because of this, you mustn't try to empty a list (or a range within a list) by removing items at monotonically increasing indices. You should either start with the highest index and move towards the head of the list, or remove at the same index (the lowest in the range) some number of times. As an example of the latter, the following code removes the first five items in the list:

```
for ( int32 i = 0; i < 5; i++ )
   myList->RemoveItem(0);
```

See also: `MakeEmpty()`

SortItems()

void ***SortItems(**int (**compareFunc*)(const void *, const void *)**)**

Rearranges the items in the list. The items are sorted using the *compareFunc* comparison function passed as an argument. This function should return a negative number if the first item is ordered before the second, a positive number if the second is ordered before the first, and 0 if the two items are ordered equivalently.

The arguments passed to the comparison function are declared to be `void*`; however, they should be regarded as pointers to the items in the list—in other words, as pointers to pointers.

Operators

= (assignment)

```
BList& operator =(const BList&)
```

Copies the contents of one BList object into another:

```
BList newList = oldList;
```

After the assignment, each object has its own independent copy of list data; destroying one of the objects won't affect the other.

Only the items in the list are copied, not the data they point to.

The Be Book The Support Kit The Support Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BList Function Summary

Function Summary

Function	Description
=	Copies the contents of one list to another
AddItem()	Adds an item to the list
AddList()	Adds contents of a list to the list
CountItems()	Returns the number of items in the list
DoForEach()	Calls a function on each item in the list
FirstItem()	Returns the first item in the list
HasItem()	Tests whether the item is in the list
IndexOf()	Returns the index of a given item in the list
IsEmpty()	Tests whether the list is empty
ItemAt()	Returns the item at a given position in the list
Items()	Returns a pointer to the list
LastItem()	Returns the last item in the list
BList()	Constructor
~BList()	Destructor
MakeEmpty()	Empties the list
RemoveItem()	Remove a given item from the list
RemoveItems()	Remove a run of items from the list
SortItems()	Orders items in the list

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BInputServerMethod Function Summary

Function Summary

Function	Description
EnqueueMessage()	Insert an event into the Input Server's queue.
BInputServerMethod()	Constructor
~BInputServerMethod()	Destructor
MethodActivated()	Input method activation hook.
SetIcon()	Set the method's icon.
SetMenu()	Assign a menu to the input method.
SetName()	Set the method's name.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

System Information

Declared in: be/kernel/OS.h
Library: libroot.so
Summary: *more...*

The following functions, types, and structures convey information about the system, such as the number of CPUs, when the kernel was built, and whether your computer is on fire.

System Info Functions

get_system_info()

```
status_t get_system_info(system_info *info)
```

The `get_system_info()` function tells you more than you want to know about the physical capacities of your computer and other statistics of your operating system. The function copies this information into the `system_info` argument *info*. You must allocate *info* before passing it in.

RETURN CODES

`get_system_info()` always returns

- B_OK.

is_computer_on()

```
int32 is_computer_on(void)
```

Returns 1 if the computer is on. If the computer isn't on, the value returned by this function is undefined.

is_computer_on_fire()

```
double is_computer_on_fire(void)
```

Returns the temperature of the motherboard if the computer is currently on fire. If the computer isn't on fire, the function returns some other value.

System Info Structures and Constants

cpu_info

```
typedef struct {
    bigtime_t active_time;
    } cpu_info
```

The `cpu_info` structure provides information about your computer's CPU(s). You retrieve the structure by looking in the **cpu_infos** field of the `system_info` structure. The **cpu_infos** field is an array that contains one `cpu_info` structure for each CPU on your motherboard.

Field	Meaning
active_time	The number of microseconds the CPU has spent doing useful work (i.e. not busy waiting) since the machine was booted.

cpu_type

```
typedef enum cpu_types {
    B_CPU_PPC_601,
    B_CPU_PPC_603,
    B_CPU_PPC_603e,
    B_CPU_PPC_604,
    B_CPU_PPC_604e,
    B_CPU_PPC_750,
    B_CPU_PPC_686,
    B_CPU_AMD_29K,
    B_CPU_X86, Obsolete
    B_CPU_MC6502,
    B_CPU_Z80,
    B_CPU_ALPHA,
    B_CPU_MIPS,
    B_CPU_HPPA,
    B_CPU_M68K,
    B_CPU_ARM,
    B_CPU_SH,
    B_CPU_SPARC,

    B_CPU_INTEL_X86 = 0x1000,
    B_CPU_INTEL_PENTIUM,
    B_CPU_INTEL_PENTIUM75,
    B_CPU_INTEL_PENTIUM_486_OVERDRIVE,
    B_CPU_INTEL_PENTIUM_MMX,
    B_CPU_INTEL_PENTIUM_MMX_MODEL_4,
```

```
    B_CPU_INTEL_PENTIUM_MMX_MODEL_8,
    B_CPU_INTEL_PENTIUM75_486_OVERDRIVE,
    B_CPU_INTEL_PENTIUM_PRO,
    B_CPU_INTEL_PENTIUM_II,
    B_CPU_INTEL_PENTIUM_II_MODEL_3,
    B_CPU_INTEL_PENTIUM_II_MODEL_5,
    B_CPU_INTEL_CELERON,
    B_CPU_INTEL_PENTIUM_III,

    B_CPU_AMD_X86 = 0x1100,
    B_CPU_AMD_K5_MODEL0,
    B_CPU_AMD_K5_MODEL1,
    B_CPU_AMD_K5_MODEL2,
    B_CPU_AMD_K5_MODEL3,
    B_CPU_AMD_K6_MODEL6,
    B_CPU_AMD_K6_MODEL7,
    B_CPU_AMD_K6_MODEL8,
    B_CPU_AMD_K6_2,
    B_CPU_AMD_K6_MODEL9,
    B_CPU_AMD_K6_III,
    B_CPU_AMD_ATHLON_MODEL1,

    B_CPU_CYRIX_X86 = 0x1200,
    B_CPU_CYRIX_GXm,
    B_CPU_CYRIX_6x86MX,

    B_CPU_IDT_X86 = 0x1300,
    B_CPU_IDT_WINCHIP_C6,
    B_CPU_IDT_WINCHIP_2,

    B_CPU_RISE_X86 = 0x1400,
    B_CPU_RISE_mP6
        } cpu_type;
```

The **B_X86_CPU** constant is obsolete. All x86 CPUs are represented by specific vendor/model constants.

These constants represents different CPU vendors and models. To retrieve the constant that represents the CPU(s) that your computer uses, look in the **cpu_type** field of the **system_info** structure.

Note that the x86 types are grouped by vendor, where the vendor is identified by the high word, and the model by the low word. Each vendor has its own **cpu_type** constant, as indicated by the explicit values in the list above. If you're only interested in the vendor, mask the **cpu_type** value with **B_CPU_X86_VENDOR_MASK**, thus:

```
system_info sysinfo;
```

```
get_system_info( &sysinfo );

switch ( sysinfo.cpu_type & B_CPU_X86_VENDOR_MASK ) {
   case B_CPU_INTEL_X86: ...
   case B_CPU_AMD_X86: ...
   case B_CPU_CYRIX_x86: ...
   case B_CPU_IDT_X86: ...
   case B_CPU_RISE_X86: ...
   default:
   /* Not x86. */
}
```

B_CPU_X86_VENDOR_MASK

B_CPU_X86_VENDOR_MASK

You mask your machine's **cpu_type** value with this constant to retrieve the CPU's vendor id (which is also a **cpu_type** value). If the CPU isn't an breed of x86, applying the mask will yield 0. See the **cpu_type** description for an example.

machine_id

typedef int32 **machine_id**[2]; *Currently unused*

machine_id is a 64-bit number (encoded as two 32-bit numbers) that uniquely identifies this specific computer. The id number is returned as the **id** field (*currently unused*) of the **system_info** structure.

B_MAX_CPU_COUNT

B_MAX_CPU_COUNT

This constant is set to the maximum number of CPUs that the BeOS can take advantage of. The number is machine-dependent.

platform_type

typedef enum `platform_types` {
    B_BEBOX_PLATFORM = 0,
    B_MAC_PLATFORM,
    B_AT_CLONE_PLATFORM,
    B_ENIAC_PLATFORM,
    B_APPLE_II_PLATFORM,

```
    B_CRAY_PLATFORM,
    B_LISA_PLATFORM,
    B_TI_994A_PLATFORM,
    B_TIMEX_SINCLAIR_PLATFORM,
    B_ORAC_1_PLATFORM,
    B_HAL_PLATFORM,
    B_BESM_6_PLATFORM,
    B_MK_61_PLATFORM,
    B_NINTENDO_64_PLATFORM
        } platform_type;
```

These constants represent the various computer platforms that the BeOS does, has, should, might, or might not run on. To retrieve the constant for the machine that you're running on, look in the **platform_type** field of the **system_info** structure.

---

system_info

```
typedef struct {
    machine_id id; Currently unused
    bigtime_t boot_time;
    int32 cpu_count;
    cpu_type cpu_type;
    int32 cpu_revision;
    cpu_info cpu_infos[B_MAX_CPU_NUM];
    int64 cpu_clock_speed;
    int64 bus_clock_speed;
    platform_type platform_type;
    int32 max_pages;
    int32 used_pages;
    int32 page_faults;
    int32 max_sems;
    int32 used_sems;
    int32 max_ports;
    int32 used_ports;
    int32 max_threads;
    int32 used_threads;
    int32 max_teams;
    int32 used_teams;
    char kernel_name[B_FILE_NAME_LENGTH];
    char kernel_build_date[B_OS_NAME_LENGTH];
    char kernel_build_time[B_OS_NAME_LENGTH];
    int64 kernel_version;
    } system_info;
```

The **system_info** structure describes your computer's hardware and operating system, and provides information about the availability of kernel resources (such as threads and ports).

You retrieve a system_info structure through the **get_system_info()** function.

Field	Meaning
id	A 64-bit number (encoded as two **int32**s) that uniquely identifies this machin
boot_time	The time at which the computer was last booted, measured in microseconds can also get this information through **system_time()**.
cpu_count	The number of CPUs on your computer's motherboard.
cpu_type	A constant that represents the CPU(s) make and model.
cpu_revision	The revision number of the CPU(s).
cpu_infos	An array of **cpu_info** structures, one for each CPU.
cpu_clock_speed	The speed (in Hz) at which the CPU(s) operate.
bus_clock_speed	The speed (in Hz) at which the bus operates.
platform_type	The platform type constant that reprsents your computer.
max_resource **used_resource**	These fields give the total number of RAM pages, semaphores, and so on, th the number that are currently in use.
page_faults	The number of times the system has read a page of memory into RAM due t
kernel_name	The filename of the kernel (the leaf name, not the path).
kernel_build_date	Fixed-width string that gives the date that the kernel was built; for example: `Jun 5 1999` **kernel_build_time** Fixed-width string that gives the time of day that the 10:02:16 **kernel_version** A Be-assigned number that identifies the kernel version.

The Be Book The Kernel Kit The Kernel Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

System Information Function Summary

Function Summary

Function	Description
get_system_info()	Returns information about the physical characteristics of the system
is_computer_on()	Tests whether the computer is on
is_computer_on_fire()	Returns the temperature of the motherboard if the computer is on fire

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Input Server Structures and Constants

Structures

input_device_ref

Declared in: be/add-ons/input_server/InputServerDevice.h

```
struct input_device_ref {
    char *name;
    input_device_type type;
    void *cookie;
    }
```

The `input_device_ref` structure is used to identify specific devices that a BInputServerDevice manages. To tell the Input Server about the devices your object manages, you create an array of these structures and pass them in a `BInputServerDevice::RegisterDevices()` call.

The fields are:

Field	Meaning
const char *`name`	An arbitrary—but preferably human-readable—name that you invent. Although the name is used as an identifier (in `Control()` calls), you should choose a name that's "UI-suitable" (e.g. "ADB Mouse", or "USB Keyboard").
input_device_type `type`	This is either `B_POINTING_DEVICE` (i.e. mice), `B_KEYBOARD_DEVICE`, or `B_UNDEFINED_DEVICE`, where the last of these is a catchall for anything that isn't a mouse or a keyboard.
void *`cookie`	This is an arbitrary piece of data that you can associate with the device. Use it to conveniently store some cogent piece of data (such as *`this`), or to more specifically identify the device (such as the pathname of the driver it corresponds to), and so on.

Constants

Device Types

Declared in: be/interface/Input.h

```
B_POINTING_DEVICE
B_KEYBOARD_DEVICE
B_UNDEFINED_DEVICE
```

These constants represent the different types of input devices; they're used when defining an `input_device_ref` structure, and when sending control messages through a BInputDevice object.

- `B_POINTING_DEVICE` signifies devices, such as mice and tablets, that are used to control the cursor.
- `B_KEYBOARD_DEVICE` signifies keyboards.
- `B_UNDEFINED_DEVICE` is anything that isn't a pointer or keyboard.

Note that there is no "joystick device"—the Input Server doesn't currently handle joysticks.

Input , Method , Operations

Declared in: be/interface/Input.h

```
enum input_method_op {
    B_INPUT_METHOD_STARTED,
    B_INPUT_METHOD_STOPPED,
    B_INPUT_METHOD_CHANGED,
    B_INPUT_METHOD_LOCATION_REQUEST
        }
```

Input , Device , Notifications

Declared in: be/interface/Input.h

```
enum input_device_notification {
    B_INPUT_DEVICE_ADDED,
    B_INPUT_DEVICE_STARTED,
    B_INPUT_DEVICE_STOPPED,
    B_INPUT_DEVICE_REMOVED
        }
```

Input , Device , Control , Messages

Declared in: be/add-ons/input_server/InputServerDevice.h

B_KEY_MAP_CHANGED
B_KEY_LOCKS_CHANGED
B_KEY_REPEAT_DELAY_CHANGED
B_KEY_REPEAT_RATE_CHANGED
B_MOUSE_TYPE_CHANGED
B_MOUSE_MAP_CHANGED
B_MOUSE_SPEED_CHANGED
B_CLICK_SPEED_CHANGED

These constants are used in the **BInputServerDevice::Control()** function to tell a BInputServerDevice that a change to a device parameter has been requested.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BWindow

Derived from: BLooper > BHandler
Mix-in classes: BArchivable
Declared in: be/interface/Window.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BWindow object represents a window that can be displayed on the screen, and that can be the target of user events. You almost always create your own BWindow subclass(es) rather than use direct instances of BWindow.

BWindow objects draw windows by talking to the App Server. If you want to take over the entire screen or draw directly into the graphics card's frame buffer (by-passing the App Server), you should use a BDirectWindow or BWindowScreen object (both classes are defined in the Game Kit).

Creating and Using a BWindow

You must create your BApplication object (**be_app**) before you create any windows. **be_app** needn't be running to construct—or even show—a window, but it must be running for the window to receive notifications of user events (mouse clicks, key presses, etc.).

Typically, the first thing you do with your BWindow is add BViews to it, through the **AddChild()** function. Again, **be_app** needn't be running at this point, nor must the window be showing.

Even though it inherits from BLooper, you never invoke **Run()** on a BWindow. Instead, you call **Show()**. In addition to putting the window on-screen, the first (and *only* the first) **Show()** invocation starts the BWindow's message loop. To remove a window from the screen without interrupting the object's message loop, use **Hide()**. Other message loop details (locking and quitting in particular) are handled as described in the BLooper class.

If you create a BWindow-derived class that uses multiple inheritance, make sure the first class your mixin class inherits from is BWindow; otherwise, you'll crash when you try to close the window. This happens because of an interaction between the window thread how C++ deletes objects of a multiply-inherited class. In other words:

```
class myClass : public BWindow, public OtherClass
```

is safe, while

```
class myClass : public OtherClass, public BWindow
```

is not.

Hook Functions

- FrameMoved()
- FrameResized()
- MenusBeginning()
- MenusEnded()
- **Minimize()**
- ScreenChanged()
- WindowActivated()
- WorkspaceActivated()
- WorkspacesChanged()
- Zoom()

Constructor and Destructor

BWindow()

```
BWindow(BRect frame,
    const char *title,
    window_type type,
    uint32 flags,
    uint32 workspaces = B_CURRENT_WORKSPACE)
BWindow(BRect frame,
    const char *title,
    window_look look,
    window_feel feel,
    uint32 flags,
    uint32 workspace = B_CURRENT_WORKSPACE)
BWindow(BMessage *archive)
```

Creates a new window:

- *frame* is the window's *content area* rectangle given in screen coordinates. The window's border and title tab (if any) are wrapped *around* the content area. The frame's coordinates are rounded to the nearest whole number. You can further restrict the window's frame through `SetWindowAlignment()` and `SetSizeLimits()`.
- *title* is the string that's displayed in the window's title tab (if it has one). The string is also used, with the prefix "w>", as the name of the window's looper thread: "w>*title*". You can reset the title through `SetTitle()`.
- *look*/*feel* or *type*. The appearance and behavior of the window is defined by a combination of the *look* and *feel* arguments, or by a single *type*. See the `window_look`, `window_feel`, and `window_type` descriptions for a list of values. The look and feel can be reset through `SetLook()`, `SetFeel()`, and `SetType()`.
- The *flags* argument is a mask that defines the window's "user attributes"—whether the user can resize or close the window, whether the window avoids front when the user closes other windows, and so on. See "Window Flags." for a list of values. The flags can be reset through `SetFlags()`.
- *workspaces* is a mask that tells the window which of the 32 potential workspace(s) it should be displayed in (it can be displayed in more than one workspace at the same time). For example, a *workspaces* of (1<<2)+(1<<7) tells the window to display itself in workspaces 2 and 7. Instead of passing a bitmask, you can also use the `B_CURRENT_WORKSPACE` or `B_ALL_WORKSPACES` constant. By default, the window appears in the current (active) workspace. You can reset the workspaces through `SetWorkspaces()`.

A freshly-created window is hidden and locked. After construction, you add BViews to the BWindow through `AddChild()`, and then show the window (and start its message loop) by

calling **Show()**.

~BWindow()

You never delete your BWindows; call **Quit()** instead.

Static Functions

- Instantiate() see BArchivable::Instantiate()

Member Functions

Activate() , IsActive() , WindowActivated()

void **Activate(**bool *flag* = **true)**
bool **IsActive(**void**)** const
virtual void **WindowActivated(**bool *active***)**

Activate() makes the BWindow the active window (if *flag* is **true**), or causes it to relinquish that status (if *flag* is **false**). The active window is made frontmost, its title tab is highlighted, and it becomes the target of keyboard events. The **Show()** function automatically activates the window.

You can't activate a hidden window.

You rarely call **Activate()** yourself. Deciding which window to make active is the user's choice.

IsActive() returns **true** if the window is currently the active window, and **false** if it's not.

WindowActivated() is a hook function that's automatically invoked on the BWindow (and each of its BView children) when the window is activated or deactivated. If *active* is **true**, the window has just become the active window; if it's **false**, it's about to give up that status. You can implement **WindowActivated()** to do whatever you want; the default implementation does nothing.

See also: **BView::WindowActivated()**

AddChild() , RemoveChild() , ChildAt() , CountChildren()

void **AddChild(**BView **aView*, BView **sibling* = NULL**)**
bool **RemoveChild(**BView **aView***)**
BView ***ChildAt(**int32 *index***)** const
int32 **CountChildren(**void**)** const

AddChild() places *aView* (and its child views) in the window, adds it to the window's view list, and adds it to the window's list of handlers:

- Graphically, the view is placed in the window's coordinate system at the location defined by its the view's frame rectangle.
- In the window's view list, *aView* is inserted before *sibling*. If *sibling* is **NULL**, *aView* is added at the end of the list. Note, however, that window list order is of little significance; for example, it doesn't affect the order in which sibling views are drawn.

- *aView* and its children are added to the window's *handler list*; *aView*'s *next handler* is set to this BWindow.

Each BView in *aView*'s hierarchy is sent an **AttachedToWindow()** call. When they've all had a chance to respond, they're each sent an **AllAttached()** call.

If *aView* has already been added to a view hierarchy, or if *sibling* isn't in the window's view list, **AddChild()** fails.

RemoveChild() removes *aView* (and its children) from the window's display, view list, and handler list, and sets *aView*'s next handler to **NULL**. **DetachedFromWindow()** and **AllDetached()** are invoked on *aView* and each of its children. If *aView* isn't in the window's view list, the function fails and returns **false**; it returns **true** upon success.

ChildAt() returns the *index*'th view in the window's view list, or **NULL** if *index* is out of bounds (you've reached the end of the list). The view list doesn't recurse; to build a full map of a window's view hierarchy, you call **BView::ChildAt()** iteratively on each of the window's views (and each of their children, etc.).

See also: **BView::Parent()**

AddShortcut() , RemoveShortcut()

void **AddShortcut(**uint32 *key*, uint32 *modifiers*, BMessage **message***message***)** void **AddShortcut(**uint32 *key*, uint32 *modifiers*, BMessage **message*, BHandler **handler***handler***)**
void **RemoveShortcut(**uint32 *key*, uint32 *modifiers***)**

AddShortcut() creates a *keyboard shortcut*: When the user types **Command**+*modifiers*+*key*, *message* is sent to *handler*. If a shortcut already exists for *modifiers*+*key*, it's removed before the new shortcut is added.

RemoveShortcut() removes the shortcut for *modifiers*+*key*.

Don't use these functions to create and remove menu shortcuts; use BMenuItem objects instead.

Notes on the arguments:

- *key* is a case-insensitive character value. If you want to map to an uppercase character, you have to include **B_SHIFT_KEY** in the *modifiers* mask.
- *modifiers* is an OR'd list of modifier key numbers. **B_COMMAND_KEY**, which is always assumed, needn't be added to the mask. See **modifiers()** for a list of modifier keys.

- *message* is a model of the BMessage you want sent when the user types the shortcut. The BWindow takes ownership of the *message* object and adds a "when" field to it:

Field name	Type code	Description
"when"	B_INT64_TYPE	The time of the key-down, in microseconds since 01/01/70.

- *handler* must be in the window's *handler list* (or the message won't be sent). If you exclude the argument, this BWindow handles the message. If *handler* is (literally) **NULL**, the message is sent to the BWindow's *focus view* (or to the BWindow if no view is in focus).

As with all **Command** events, a B_KEY_DOWN message isn't sent when the user invokes a keyboard shortcut, but the subsequent B_KEY_UP is.

Every BWindow has five built-in shortcuts:

Shortcut	Message	Handler
Command+**x**	B_CUT	the focus view
Command+**c**	B_COPY	the focus view
Command+**v**	B_PASTE	the focus view
Command+**a**	B_SELECT_ALL	the focus view
Command+**w** ***(closable windows only)***	B_QUIT_REQUESTED	the BWindow

In addition, BWindows respond to **Command**+**q** by posting B_QUIT_REQUESTED to be_app.

AddToSubset() , RemoveFromSubset()

status_t **AddToSubset(**BWindow **window***)**
status_t **RemoveFromSubset(**BWindow **window***)**

Adds windows to and removes them from this window's subset. This affects modal and floating windows with a subset feel only (i.e. B_MODAL_SUBSET_WINDOW_FEEL or B_FLOATING_SUBSET_WINDOW_FEEL). A subset feel window blocks or floats above only those windows in its subset. To set the window's feel, use SetFeel().

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Archives the BWindow by recording its frame rectangle, title, type, and flags in the BMessage *archive*. If the *deep* flag is `true`, this function also archives all the views in the window's view hierarchy. If the flag is `false`, only the BWindow is archived.

See also: BArchivable::Archive(), Instantiate() static function

---

Bounds() , Frame()

```
BRect Bounds(void) const
BRect Frame(void) const
```

These functions return the rectangle that encloses the window's content area. The *bounds rectangle* (**Bounds()**) is expressed in the window's coordinate system; the *frame rectangle* (**Frame()**) is expressed in the screen's coordinate system.

The rectangles are cached by the BWindow itself—calling these functions doesn't incur a trip to the App Server.

ChildAt() see AddChild()

CountChildren() see AddChild()

CommitViewTransaction() see OpenViewTransaction()

ConvertToScreen() , ConvertFromScreen()

```
BPoint ConvertToScreen(BPoint windowPoint) const
void ConvertToScreen(BPoint *windowPoint) const
BRect ConvertToScreen(BRect windowRect) const
void ConvertToScreen(BRect *windowRect) const
BPoint ConvertFromScreen(BPoint screenPoint) const
void ConvertFromScreen(BPoint *screenPoint) const
BRect ConvertFromScreen(BRect screenRect) const
void ConvertFromScreen(BRect *screenRect) const
```

Converts the argument from window coordinates to screen coordinates or vice versa. The point or rect needn't fall within this BWindow's bounds.

If the argument is passed by value, the function returns the converted value; if it's by pointer, the conversion is done in-place.

The BWindow must be locked.

CurrentFocus()

```
BView *CurrentFocus(void) const
```

Returns the current focus view for the BWindow, or **NULL** if no view is currently in focus. The focus view is the BView that's responsible for showing the current selection and is the target for keyboard messages directed at this BWindow. The focus view is set through **BView::MakeFocus()**.

The BWindow sets its preferred handler to be the focus view, so the inherited **PreferredHandler()** function will return this same object (but as a BHandler).

DefaultButton() see SetDefaultButton()

DisableUpdates() , EnableUpdates()

void **DisableUpdates**(void)
void **EnableUpdates**(void)

These functions disable automatic updating within the window, and re-enable it again. Any drawing that's done while updates are disabled is suppressed until updates are re-enabled. If you're doing a lot of drawing within the window, and you want the results of the drawing to appear all at once, you should disable updates, draw, and then re-enable updates.

See also: **BView::Invalidate()**, **UpdateIfNeeded()**

DispatchMessage()

Implementation detail; see **BLooper::DispatchMessage()**.

You shouldn't override this function in a BWindow subclass; if you want to augment the window's message-dispatching mechanism, override **MessageReceived()**.

EnableUpdates() see DisableUpdates()

Feel() see SetFeel()

Flags() see SetFlags()

FindView()

BView ***FindView**(BPoint *point*) const
BView ***FindView**(const char **name*) const

Returns the view located at *point* within the window, or the view tagged with *name*. The functions returns **NULL** if no view is found.

Flush() , Sync()

void **Flush**(void) const
void **Sync**(void) const

Both of these functions cause this window's App Server-bound messages to be sent immediately. `Flush()` sends the messages and returns immediately; `Sync()` send the messages and waits for the App Server to respond. In other words, when `Sync()` returns you're guaranteed that all of the flushed messages have been processed.

See also: `BView::Flush()`

- Frame() see Bounds()

- FrameMoved() , FrameResized()

virtual void **FrameMoved**(BPoint *origin*)
virtual void **FrameResized**(float *width*, float *height*)

These hook functions are invoked just after the window's frame is moved or resized, whether by the user or programatically. The arguments give the window's new origin (in screen coordinates) or dimensions. The default implementations do nothing.

See also: `B_WINDOW_MOVED`, `B_WINDOW_RESIZED`.

- GetSizeLimits() see SetSizeLimits()

- GetSupportedSuites()

virtual status_t **GetSupportedSuites**(BMessage **message*)

Adds the name "suite/vnd.Be-window" to the *message*.

See also: `BHandler::GetSupportedSuites()`

- GetWindowAlignment() see SetWindowAlignment()
- Hide() see Show()
- IsFloating() see IsFront()

- IsFront() , IsFloating() , IsModal()

bool **IsFront**(void) const
bool **IsFloating**(void) const
bool **IsModal**(void) const

These functions return **true** if the window is frontmost on screen, if it has a floating window feel, and if it has a modal window feel, respectively.

A floating window can never be the frontmost window.

- IsMinimized() see Show()
- IsModal() see IsFront()
- KeyMenuBar() see SetKeyMenuBar()

- LastMouseMovedView()

BView ***LastMouseMovedView**(void)*

Returns a pointer to the view in this window that most recently received a **B_MOUSE_MOVED** message.

- Look() see SetLook()

- MenusBeginning() , MenusEnded()

virtual void **MenusBeginning**(void)
virtual void **MenusEnded**(void)

The **MenusBeginning()** hook function is called just before menus belonging to the window are about to be shown to the user. **MenusEnded()** is called when the menus have been removed from the screen. The default implementations do nothing. You can implement these functions to make sure the menus' states accurately reflect the state of the window.

These hook functions are not invoked because of messages—don't go looking for a "menus beginning" or "menus ended" message.

- MessageReceived()

virtual void **MessageReceived**(BMessage **message*)

Implementation detail. See **BHandler::MessageReceived()**.

- Minimize() see Show()

MoveBy() , MoveTo()

void **MoveBy(**float *horizontal*, float *vertical***)**
void **MoveTo(**BPoint *point***)** void **MoveTo(**float *x*, float *y***)**

These functions move the window without resizing it. **MoveBy()** adds *horizontal* coordinate units to the left and right components of the window's frame rectangle and *vertical* units to the frame's top and bottom. If *horizontal* and *vertical* are negative, the window moves upward and to the left. If they're positive, it moves downward and to the right. **MoveTo()** moves the left top corner of the window's content area to *point*—or (*x*, *y*)—in the screen coordinate system; it adjusts all coordinates in the frame rectangle accordingly.

None of the values passed to these functions should specify fractional coordinates; a window must be aligned on screen pixels. Fractional values will be rounded to the closest whole number.

Neither function alters the BWindow's coordinate system or bounds rectangle.

When these functions move a window, a window-moved event is reported to the window. This results in the BWindow's **FrameMoved()** function being called.

See also: **FrameMoved()**

NeedsUpdate()

bool **NeedsUpdate(**void**)** const

Returns `true` if any of the views within the window need to be updated, and `false` if they're all up-to-date.

OpenViewTransaction() , CommitViewTransaction()

virtual void **OpenViewTransaction(**void**)**
virtual void **CommitViewTransaction(**void**)**

These two functions bracket a series of "batched" drawing instructions. After you call **OpenViewTransaction()**, everything you draw in the window is bundled up and then sent to the app server (and rendered) when you call **CommitViewTransaction()**. You can only perform one transaction (per window) at a time. The BWindow must be locked when you call **OpenViewTransaction()**, and must remain locked until after you call **CommitViewTransaction()**. Invocations of **Flush()** are ignored while a transaction is open (and locked).

PreferredHandler() see CurrentFocus()

PulseRate() see SetPulseRate()

Quit()

virtual void **Quit**(void)

Quit() removes the window from the screen, deletes all the BViews in its view hierarchy, destroys the window thread, removes the window's connection to the Application Server, and deletes the BWindow object.

Use this function, rather than the `delete` operator, to destroy a window.

BWindow's **Quit()** works much like the BLooper function it overrides. When called from the BWindow's thread, it doesn't return. When called from another thread, it returns after all previously posted messages have been processed and the BWindow and its thread have been destroyed.

The window must be locked when you call **Quit()**.

See also: **BLooper::QuitRequested()**, **BLooper::Quit()**, **BApplication::QuitRequested()**

RemoveChild() see AddChild()

RemoveShortcut() see AddShortcut()

ResizeBy() , ResizeTo()

void **ResizeBy**(float *horizontal*, float *vertical*)
void **ResizeTo**(float *width*, float *height*)

These functions resize the window, while keeping its left top corner constant. **ResizeBy()** adds *horizontal* pixels to the width of the window's frame and *vertical* pixels to its height. **ResizeTo()** sets the frame absolutely to [*width*, *height*] pixels. Fractional components are rounded to the nearest whole number.

The **FrameResized()** hook function is called after the frame has been resized.

ResolveSpecifier()

virtual BHandler ***ResolveSpecifier**(BMessage **message*, int32 *index*, BMessage **specifier*, int32 *command*, const char **property*)

Resolves specifiers for the "Frame", "Title", and "View" properties. See "Scripting Support" in the class overview for more information.

See also: `BHandler::ResolveSpecifier()`

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Hook function that's called when the screen (on which this window is located) changes size or location in the *screen coordinate system*, or changes color space (depth). *frame* is the screen's new frame rectangle, and *mode* is its new color space.

See also: `BScreen::Frame()`

SendBehind()

status_t **SendBehind(**const BWindow ***window***)

Relayers the windows on the screen so this window is behind *window*.

SetDefaultButton() , DefaultButton()

void **SetDefaultButton(**BButton ***button***)
BButton ***DefaultButton(**void**)** const

Set and return the window's *default button*. This is the button that's mapped to the **Enter** key. The user can activate the default button at any time—even if another BView is the focus view (the focus view will *not* receive a `B_KEY_DOWN` message). To remove the current default (without promoting another button) call `SetDefaultButton(NULL)`. There can only be one default button at a time; `SetDefaultButton()` demotes the previous default.

When you promote or demote a default button, it's automatically redisplayed and receives a `BButton::MakeDefault()` call.

SetFeel() , Feel()

status_t **SetFeel**(window_feel *feel*)
window_feel **Feel(**void**)** const

`SetFeel()` changes the window's *feel* to the specified value.

`Feel()` returns the current feel of the window.

See the BWindow constructor for a list of `window_feel` constants.

SetFlags() , Flags()

status_t **SetFlags**(uint32 *flags*)
uint32 **Flags**(void) const

SetFlags() set the window's flags (or "user attributes") to the specified combination. **Flags()** returns the current flags.

See "Window Flags." for a list of the flag values.

SetKeyMenuBar() , KeyMenuBar()

void **SetKeyMenuBar**(BMenuBar **menuBar*)
BMenuBar ***KeyMenuBar**(void) const

SetKeyMenuBar() makes the specified BMenuBar object the "key" menu bar for the window—the object that's at the root of the menu hierarchy that users can navigate using the keyboard. **KeyMenuBar()** returns the object with key status, or `NULL` if the window doesn't have a BMenuBar object in its view hierarchy.

If a window contains only one BMenuBar view, it's automatically designated the key menu bar. If there's more than one BMenuBar in the window, the last one added to the window's view hierarchy is considered to be the key one.

If there's a "true" menu bar displayed along the top of the window, its menu hierarchy is the one that users should be able to navigate with the keyboard. **SetKeyMenuBar()** can be called to make sure that the BMenuBar object at the root of that hierarchy is the "key" menu bar.

SetLook() , Look()

status_t **SetLook**(window_look *look*)
window_look **Look**(void) const

`SetLook()` changes the window's *look* to the specified value.

`Look()` returns the current look of the window.

See the BWindow constructor for a list of **window_look** constants.

SetPulseRate() , PulseRate()

void **SetPulseRate**(bigtime_t *microseconds*)
bigtime_t **PulseRate**(void)

These functions set and return how often **Pulse()** is called for the BWindow's views (how often **B_PULSE** messages are posted to the window). All BViews attached to the same window share the same pulse rate.

By turning on the **B_PULSE_NEEDED** flag, a BView can request periodic **Pulse()** notifications. By default, **B_PULSE** messages are posted every 500,000 microseconds, as long as no other messages are pending. Each message causes **Pulse()** to be called once for every BView that requested the notification. There are no pulses if no BViews request them.

SetPulseRate() permits you to set a different interval. The interval set should not be less than 100,000 microseconds; differences less than 50,000 microseconds may not be noticeable. A finer granularity can't be guaranteed.

Setting the pulse rate to 0 disables pulsing for all views in the window.

See also: **BView::Pulse()**, the BView constructor

SetSizeLimits() , GetSizeLimits() , SetZoomLimits()

void **SetSizeLimits(**float *minWidth*, float *maxWidth*, float *minHeight*, float *maxHeight***)**
void **GetSizeLimits(**float **minWidth*, float **maxWidth*, float **minHeight*, float **maxHeight***)**
void **SetZoomLimits(**float *maxWidth*, float *maxHeight***)**

These functions set and report limits on the size of the window. The user won't be able to resize the window beyond the limits set by **SetSizeLimits()**—to make it have a width less than *minWidth* or greater than *maxWidth*, nor a height less than *minHeight* or greater than *maxHeight*. By default, the minimums are sufficiently small and the maximums sufficiently large to accommodate any window within reason.

SetSizeLimits() constrains the user, not the programmer. It's legal for an application to set a window size that falls outside the permitted range. The limits are imposed only when the user attempts to resize the window; at that time, the window will jump to a size that's within range.

GetSizeLimits() writes the current limits to the variables provided.

SetZoomLimits() sets the maximum size that the window will zoom to (when the **Zoom()** function is called). The maximums set by **SetSizeLimits()** also apply to zooming; the window will zoom to the screen size or to the smaller of the maximums set by these two functions.

Since the sides of a window must line up on screen pixels, the values passed to both **SetSizeLimits()** and **SetZoomLimits()** should be whole numbers.

See also: the BWindow constructor, **Zoom()**

SetTitle() , Title()

void **SetTitle(**const char **newTitle***)*
const char ***Title(**void**)** const

These functions set and return the window's title. **SetTitle()** replaces the current title with *newTitle*. It also renames the window thread in the following format:

"w>*newTitle*"

where as many characters of the *newTitle* are included in the thread name as will fit.

Title() returns a pointer to the current title. The returned string is null-terminated. It belongs to the BWindow object, which may alter the string or free the memory where it resides without notice. Applications should ask for the title each time it's needed and make a copy for their own purposes.

A window's title and thread name are originally set by an argument passed to the BWindow constructor.

See also: the BWindow constructor

SetType() , Type()

status_t **SetType**(window_type *type*)
window_type **Type(**void**)** const

`SetType()` changes the type of the window to the specified value. `Type()` returns the type. You normally set the window's type when it's constructed. .

The type is set at construction (or by `SetType()`) as one of the following constants (full descriptions can be found in the discussion of the BWindow constructor):

B_UNTYPED_WINDOW
B_MODAL_WINDOW
B_BORDERED_WINDOW
B_TITLED_WINDOW
B_DOCUMENT_WINDOW
B_FLOATING_WINDOW

See also: the BWindow constructor

SetWindowAlignment() , GetWindowAlignment()

status_t **SetWindowAlignment(**window_alignment *mode*, int32 *h*, int32 *hOffset* = 0, int32 *width* = 0,

```
    int32 widthOffset = 0,
    int32 v = 0,
    int32 vOffset = 0,
    int32 height = 0,
    int32 heightOffset = 0)
status_t GetWindowAlignment(window_alignment *mode = NULL,
    int32 *h = NULL,
    int32 *hOffset = NULL,
    int32 *width = NULL,
    int32 *widthOffset = NULL,
    int32 *v = NULL,
    int32 *vOffset = NULL,
    int32 *height = NULL,
    int32 *heightOffset = NULL) const
```

`SetWindowAlignment()` sets the current alignment of the window content on the screen. *mode* is either `B_PIXEL_ALIGNMENT` or `B_BYTE_ALIGNMENT`.

If *mode* is `B_PIXEL_ALIGNMENT`, `SetWindowAlignment()` aligns the window in pixel coordinates. *h* and *hOffset* together determine the horizontal alignment: *h* gives the horizontal origin step while *hOffset* is the horizontal offset. *hOffset* must be between 1 and *h* (as a convenience, 0 is taken to mean 1). For example, if *h* is 4 and *hOffset* is 1, valid horizontal origins would be ..., -7, -3, 1, 5, 9, ... Similarly, *width*/*widthOffset*, *v*/*vOffset*, *height*/*heightOffset* give you control over the other window parameters.

If *mode* is `B_BYTE_ALIGNMENT`, then the alignment is given in terms of frame buffer offsets. However, the setting only affects the horizontal origin and width. You can't align the right and bottom edges in `B_BYTE_ALIGNMENT` mode.

`GetWindowAlignment()` returns the current window alignment.

Both methods return `B_NO_ERROR` on success and `B_ERROR` otherwise.

SetWorkspaces() , Workspaces()

```
void SetWorkspaces(uint32 workspaces)
uint32 Workspaces(void) const
```

These functions set and return the set of workspaces where the window can be displayed. The *workspaces* argument passed to **`SetWorkspaces()`** and the value returned by **`Workspaces()`** is a bitfield with one bit set for each workspace in which the window can appear. Usually a window appears in just one workspace.

`SetWorkspaces()` can associate a window with workspaces that don't exist yet. The window will appear in those workspaces if and when the user creates them.

You can pass **B_CURRENT_WORKSPACE** as the *workspaces* argument to place the window in the workspace that's currently displayed (the active workspace) and remove it from all others, or **B_ALL_WORKSPACES** to make sure the window shows up in all workspaces, including any new ones that the user might create. **Workspaces()** may return **B_ALL_WORKSPACES**, but will identify the current workspace rather than return **B_CURRENT_WORKSPACE**.

Changing a BWindow's set of workspaces causes it to be notified with a **WorkspacesChanged()** function call.

See also: the BWindow constructor, **WorkspacesChanged()**

- SetZoomLimits() see SetSizeLimits()
- Show() see Hide()
- Sync() see Flush()
- Title() see SetTitle()
- Type() see SetType()

- Show() , Hide() , IsHidden() , Minimize() , IsMinimized()

virtual void **Show**(void)
virtual void **Hide**(void)
bool **IsHidden**(void) const
virtual void **Minimize**(bool *minimize*)
bool **IsMinimized**(void) const

These functions hide and show the window.

Show() places the window frontmost on the screen (but behind any applicable floating or modal windows), places it on **Deskbar**'s window list, and makes it the active window. If this is the BWindow's first **Show()**, the object's message loop is started, and the object is unlocked.

Hide() removes the window from the screen, removes it from **Deskbar**'s window list, and passes active status to some other window (if this is the active window). If **Hide()** is called more than once, you'll need to call **Show()** an equal number of times for the window to become visible again.

Minimize() hides and shows the window (and passes active status), as *minimize* is **true** or **false**. The difference between this function and **Hide()**/**Show()** is that **Minimize()** dims (and undims) the window's entry in **Deskbar**'s window list, but doesn't remove the entry altogether. Also, a single **Minimize(false)** "undoes" any number of **Minimize(true)** calls.

Minimize() also acts as a hook that's invoked when the user double-clicks the window's title tab or selects the window from **DeskBar**'s window list. If *minimize* is **true**, the window is about to be hidden; if **false**, it's about to be shown. The **Minimize()** function itself does the hiding

and showing—if you override **Minimize()** and you want to inherit the BWindow behaviour, you must call **BWindow::Minimize()** in your implementation.

IsHidden() returns **true** if the window is currently hidden (i.e. through Hide()). IsMinimized() returns **true** if the window is minimized. Hiding takes precendence over minimization. For example, in both of these sequences...

```
window->Hide();
window->Minimize(true);
/* or */
window->Minimize(true);
window->Hide();
```

...the window is hidden but not minimized.

See also: B_MINIMIZE

UpdateIfNeeded()

void **UpdateIfNeeded(**void**)**

Immediately and synchronously invokes Draw() on each child view that needs updating. This function is ignored if its called from any thread other than the BWindow's message loop. You call it as part of the implementation of a user interface hook function (MouseMoved(), KeyDown(), et. al.) to force invalid views to be immediately redrawn without having to wait for the hook function to finish. See "Forcing an Update while Responding to an Event" for details and an example.

WindowActivated() see Activate()

Workspaces() see SetWorkspaces()

WorkspaceActivated()

virtual void **WorkspaceActivated(**int32 *workspace*, bool *active***)**

Implemented by derived classes to respond to a notification that the workspace displayed on the screen has changed. All windows in the newly activated workspace as well as those in the one that was just deactivated get this notification.

The *workspace* argument is an index to the workspace in question and the *active* flag conveys its current status. If *active* is `true`, the workspace has just become the active workspace. If *active* is `false`, it has just stopped being the active workspace.

The default (BWindow) version of this function is empty.

See also: "B_WORKSPACE_ACTIVATED" in the **Message Protocols** appendix, activate_workspace()

WorkspacesChanged()

virtual void **WorkspacesChanged**(uint32 *oldWorkspaces*, uint32 *newWorkspaces*)

Implemented by derived classes to respond to a notification the the window has just changed the set of workspaces in which it can be displayed from *oldWorkspaces* to *newWorkspaces*. This typically happens when the user moves a window from one workspace to another, but it may also happen when a programmatic change is made to the set of permitted workspaces. Each workspace is represented by a corresponding bit in the *oldWorkspaces* and *newWorkspaces* masks.

The default (BWindow) version of this function is empty.

See also: "B_WORKSPACES_CHANGED" in the **Message Protocols** appendix, SetWorkspaces()

Zoom()

void **Zoom**(void)
virtual void **Zoom**(BPoint *origin*, float *width*, float *height*)

The non-virtual Zoom() (which is called when the user clicks the *zoom button*, but can also be called programatically) figures out a new size and location for the window (as described below), and then passes these dimensions to the virtual Zoom() hook function. It's hook Zoom()'s responsibility to actually resize and move the window; the default version applies the arguments explicitly by calling MoveTo() and ResizeTo(). You can re-implement hook Zoom() to create a new zooming algorithm that incorporates or ignores the arguments as you see fit.

The dimensions that non-virtual Zoom() passes to hook Zoom() are deduced from the smallest of three rectangles: 1) the screen rectangle, 2) the rectangle defined by SetZoomLimits(), 3) the rectangle defined by SetSizeLimits(). However, if the window's rectangle already matches these "zoom" dimensions (give or take a few pixels), Zoom() passes the window's previous ("non-zoomed") size and location.

You can effectively call Zoom() even if the window is B_NOT_ZOOMABLE.

Zoom() may both move and resize the window, resulting in FrameMoved() and FrameResized() notifications.

Constants and Defined Types

window_look

Window Look	Meaning
B_DOCUMENT_WINDOW_LOOK	Large title bar, thick border, draggable "resize box" .
B_TITLED_WINDOW_LOOK	Same as the document window, but with a less substantial "resize co
B_FLOATING_WINDOW_LOOK	Small title bar, thin border, resize corner.
B_MODAL_WINDOW_LOOK	No title bar, thick border, no resize control (by convention; see the **B_NOT_RESIZABLE** window flag).
B_BORDERED_WINDOW_LOOK	No title bar, line border, no resize control.
B_NO_BORDER_WINDOW_LOOK	A borderless white rectangle. The user can't move or close this windo

The **window_look** constants define the appearance of the window—the look of its title tab, border, and the type of "resize control" (in the bottom right corner of the window). The look is set in the constructor or in the **SetLook()** function. The table below lists and briefly describes the window_look values.

window_feel

Window Feel	Meaning
B_NORMAL_WINDOW_FEEL	Behaves like a normal window (non-modal, non-floating).
B_MODAL_SUBSET_WINDOW_FEEL **B_MODAL_APP_WINDOW_FEEL** **B_MODAL_ALL_WINDOW_FEEL**	When displayed, blocks all windows in its subset[1], app, or acro entire system. A modal subset/app window is visible only if a w its subset/app is visible. Modal all windows are always visible i workspaces
B_FLOATING_SUBSET_WINDOW_FEEL **B_FLOATING_APP_WINDOW_FEEL** **B_FLOATING_ALL_WINDOW_FEEL**	Floats above all windows in its subset[1], app, or across the enti system[2]. A floating subset/app window is visible only if a wind subset/app is frontmost. Floating all windows are always visib workspaces.

[1] To set a window's subset, use **AddToSubset()**.

[2] Modal windows are drawn in front of floating windows.

The **window_feel** constants govern a window's behavior in relation to other windows. The feel is set in the BWindow constructor or in **SetFeel()**. The table below briefly describes the window feels.

window_type

Window Type	Look and Feel
B_TITLED_WINDOW	B_TITLED_WINDOW_LOOK and B_NORMAL_WINDOW_FEEL.
B_DOCUMENT_WINDOW	B_DOCUMENT_WINDOW_LOOK and B_NORMAL_WINDOW_FEEL.
B_MODAL_WINDOW	B_MODAL_WINDOW_LOOK and B_MODAL_APP_WINDOW_FEEL.
B_FLOATING_WINDOW	B_FLOATING_WINDOW_LOOK and B_FLOATING_APP_WINDOW_FEEL.
B_BORDERED_WINDOW	B_BORDERED_WINDOW_LOOK and B_NORMAL_WINDOW_FEEL.
B_UNTYPED_WINDOW	A window of unknown or undefined type.

The window_type constants are pre-defined combinations of looks and feels. You set the type through the BWindow constructor or SetType().

Window Flags.

Window Flag	Meaning
B_NOT_MOVABLE	Prevents the user from being able to move the window.
B_NOT_CLOSABLE	Prevents the user from closing the window.
B_NOT_ZOOMABLE	Prevents the user from zooming the window.
B_NOT_MINIMIZABLE	Prevents the user from hiding the window by double-clicking the
B_NOT_H_RESIZABLE **B_NOT_V_RESIZABLE** **B_NOT_RESIZABLE**	Prevents the user from resizing the window horizontally and/or ve (B_NOT_RESIZABLE is the same as the sum of the other two constant
B_OUTLINE_RESIZE	The window draws only the outline of its new dimensions as it's b resized, and doesn't refresh its contents.
B_WILL_ACCEPT_FIRST_CLICK	Tells a non-active window to process an activating mouse click (th mouse click) as if it were already active. The BView that responds mouse-down message must activate the window. By default, the mouse click in a non-active window activates the window, and the mouse click event is thrown away.
B_AVOID_FRONT	Prevents the window from becoming the frontmost window.
B_AVOID_FOCUS	Prevents the window from being the target of keyboard events.
B_NO_WORKSPACE_ACTIVATION	When a window is first shown, the workspace normally switches t in which the window is displayed. Setting this flag keeps this from happening.
B_NOT_ANCHORED_ON_ACTIVATE	Tells the system to bring the window to the current workspace wh window is selected from **Deskbar**'s window list. Normally, selectin window from the list activates the workspace that the window is c in.
B_ASYNCHRONOUS_CONTROLS	Tells the window to allow controls to run asynchronously. ***All* windo that contain controls should include this flag** (it's off by default of backwards compatibility).
B_QUIT_ON_WINDOW_CLOSE	Currently has no effect.

The window flags (or "user attributes") define miscellaneous aspects of the window's interface, such as whether it can be moved or closed by the user. You combine the flags that you want and pass them to the BWindow constructor or `SetFlags()`.

The default behavior is the inverse of all these flags—i.e. a window is normally movable, closable, resizable, and so on. However, some window looks and feels cause imply some of these flags. For example, a modal feel window can't be minimized (hidden)

Window Workspaces

Workspace Constant	Meaning
B_CURRENT_WORKSPACE	Appear in the current workspace.
B_ALL_WORKSPACES	Appear in all workspaces.

You tell a window which workspaces to appear through the constructor, or through the `SetWorkspaces()` function. In practice, the only two realistic settings are represented by these constants.

Scripting Support

Suite: "suite/vnd.Be-window"

B_WINDOW_MOVE_BY , B_WINDOW_MOVE_TO

Message	Meaning
B_WINDOW_MOVE_BY	Moves the window by the distance specified by the B_POINT_TYPE field "data"
B_WINDOW_MOVE_TO	Moves the window to the position specified by the B_POINT_TYPE field "data".

"Feel"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the current feel of the window.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the feel of the window.

The "Feel" property represents the workspaces in which the window resides. The messages are equivalent to manipulating the window with the **Feel()** and SetFeel() methods.

"Flags"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the current flags of the window.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the window flags.

The "Flags" property represents the window flags. The messages are equivalent to manipulating the window with the Flags() and SetFlags() methods.

"Frame"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the window's frame rectangle.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the window's frame rectangle.

The "Frame" property represents the frame rectangle of the window. The frame is passed as a BRect (B_RECT_TYPE).

"Hidden"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns `true` if the window is hidden; `false` otherwise.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Hides or shows the window.

The "Hidden" property determines the visibility of the window. The messages are equivalent to manipulating the window with the IsHidden(), Hide(), and Show() methods with one caveat: nested Hide() and Show() calls are disabled so that multiple scripted "hides" are undone with a single scripted "show".

"Look"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the current look of the window.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the look of the window.

The "Look" property represents the constant that indicates how the window appears. The messages are equivalent to manipulating the window with the **Look()** and SetLook() methods.

"MenuBar"

Message	Specifiers	Meaning
any	B_DIRECT_SPECIFIER	Directs the scripting message to the key menu bar.

The "MenuBar" property pops the current specifier off the specifier stack and then passes the scripting message to the key menu bar. If no such menu bar is present, then an error is returned.

"Minimize"

Message	Specifiers	Meaning
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Minimizes the window if "data" is `true`; restores otherwise

The "Minimize" property controls the whether the window is minimized or not. The message is equivalent to manipulating the window with the **Minimize()** method.

"Title"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns a string containing the window title.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the window title.

The "Title" property represents the title of the window. The messages are equivalent to manipulating the window with the **Title()** and SetTitle() methods.

"View"

Message	Specifiers	Meaning
any	any	Directs the scripting message to the top view without popping the current s

The "View" property redirects all requests to the window's top view without popping the specifier from the stack.

"Workspaces"

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns int32 bitfield of the workspaces in which the window
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the workspaces in which the window appears.

The "Workspaces" property represents the workspaces in which the window resides. The messages are equivalent to manipulating the window with the **Workspaces()** and SetWorkspaces() methods.

Archived Fields

Field	Type code	Meaning
"_frame"	B_RECT_TYPE	The frame rectangle.
"_title"	B_STRING_TYPE	The title of the BWindow.
"_wlook"	B_INT32_TYPE	The BWindow look.
"_wfeel"	B_INT32_TYPE	The BWindow feel.
"_type"	B_INT32_TYPE	The BWindow type (if one exists).
"_flags"	B_INT32_TYPE	The BWindow flags.
"_wspace"	B_INT32_TYPE	The workspaces in which the BWindow appears.
"_zoom" (array)	B_FLOAT_TYPE	The horizontal and vertical zoom limits, if any exist.
"_sizel" (array)	B_FLOAT_TYPE	The minimum horizontal and vertical and maximum horizontal and vertical size limits, if any exist.
"_pulse"	B_INT64_TYPE	The pulse rate, if not the default.
"_views" (array)	B_MESSAGE_TYPE	Child BViews. See Archived Fields [BView] for the BMessage format. (Deep archive only.)

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BWindow Function Summary

Function Summary

Function	Description
Activate()	Activates or deactivates the window
AddChild()	Adds a view to the window
AddShortcut()	Adds a keyboard shortcut
AddToSubset()	Adds a window to this window's subset
Archive()	Archives a BWindow into a BMessage
Bounds()	Returns the window's bounds rectangle
ChildAt()	Returns a specific view in the window
CommitViewTransaction()	Finishes and renders the current batch drawing job
ConvertFromScreen()	Converts from screen coordinates to local coordinates
ConvertToScreen()	Converts from local coordinates to screen coordinates
CountChildren()	Returns the number of "root" views in the window
CurrentFocus()	Returns the current focus view
DefaultButton()	Returns the default button for the window
DisableUpdates()	Disables automatic updating

DispatchMessage()	Implementation detail. See BLooper::DispatchMessage()
EnableUpdates()	Enables automatic updating
FindView()	Finds a specific child view
Flush()	Flushes the server queue, returning immediately
FrameMoved()	Hook called when the window is moved
FrameResized()	Hook called when the window is resized
GetSizeLimits()	Returns the maximum and minimum window size
GetSupportedSuites()	Provided for implementation
GetWindowAlignment()	Returns the window pixel alignment
Hide()	Hides a window
IsActive()	Tests whether the window is active
IsFloating()	Tests whether the window has a floating feel
IsFront()	Tests whether the window is frontmost on the screen
IsHidden()	Tests whether the window is hidden
IsMinimized()	Returns **`true`** if the window is minimized
IsModal()	Tests whether the window has a modal feel

KeyMenuBar()	Returns the key menu bar
LastMouseMovedView()	Returns a pointer to the child view in which the mouse last moved
MenusBeginning()	Hook called before menus are displayed
MenusEnded()	Hook called after menus have been removed from the screen
MessageReceived()	Implementation detail. See **`BHandler::MessageReceived()`**.
MoveBy()	Moves the window by a given amount
MoveTo()	Moves the window to a given position
NeedsUpdate()	Tests whether any views in the window need to be updated
OpenViewTransaction()	Starts a new batch drawing job
PulseRate()	Returns the pulse interval
Quit()	Removes the window and its views
RemoveChild()	Removes a view from the window
RemoveFromSubset()	Removes a window from this window's subset
RemoveShortcut()	Removes a keyboard shortcut
ResizeBy()	Resize the window by a given amount
ResizeTo()	Resize the window to the given dimensions

ResolveSpecifier()	Provided for implmentation
ScreenChanged()	Hook called when the screen characteristics change
SendBehind()	Move the window behind another window
SetDefaultButton()	Sets the default button for the window
SetFeel()	Sets the window feel
SetFlags()	Sets the window flags
SetKeyMenuBar()	Sets the key menu bar
SetLook()	Sets the window look
SetPulseRate()	Sets the pulse interval
SetSizeLimits()	Sets the maximum and minimum window size
SetTitle()	Sets the window title
SetType()	Sets the window type
SetWindowAlignment()	Sets the window pixel alignment
SetWorkspaces()	Sets the window's workspace set
SetZoomLimits()	Sets the maximum zoom size
Show()	Displays a window, returning immediately
Sync()	Flushes the server queue, returning when all operations have completed
Type()	Returns the window type
UpdateIfNeeded()	Updates dirty views. (Effective in the window thread only!)

WindowActivated()	Hook called when the window is activated or deactivated
BWindow()	Constructor
~BWindow()	Use Quit() instead.
WorkspaceActivated()	Hook called when the window's workspace is activated or deactivated
WorkspacesChanged()	Hook called when the workspace set changes
Zoom()	Zooms the window

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDirectWindow

Derived from: BWindow
Declared in: be/kit/game/DirectWindow.h
Library: libgame.so
Summary: *more...*

The BDirectWindow class gives your code direct access to the graphics frame buffer on the video card. Unlike BWindowScreen, BDirectWindow can be used in both full-screen and window modes—you can create a BDirectWindow that looks just like a normal window, but lets your code draw into it by directly accessing the frame buffer.

In addition, BDirectWindow lets you switch between full-screen exclusive mode and windowed modes without breaking down and rebuilding the object. A simple call to the `SetFullScreen()` function does the job.

If you want your direct window to be full-screen but don't want exclusive mode, just resize the window to fill the entire screen; since full-screen exclusive mode (as set by calling `SetFullScreen()`) won't let other windows draw in front of your direct window, you can't have menus in a full-screen exclusive mode direct window.

Another difference between BDirectWindow and BWindowScreen is that BDirectWindow lets you access all the BWindow functions; you can literally treat your BDirectWindow just like another window. There are two caveats:

Don't draw into the direct window from its own thread; you should spawn another thread for drawing into the direct window, and use the **`DirectConnected()`** function to synchronize the interaction between BDirectWindow and your drawing thread. Also, if you choose to use BWindow or BView API inside a BDirectWindow, be sure you don't block the `DirectConnected()` function.

Not all video cards support window mode; use the `SupportsWindowMode()` function if you need to know whether or not window mode is available.

Getting Connected (and Staying That Way)

The key to the BDirectWindow class is the `DirectConnected()` function, which your code must implement. This function is called whenever a change that your drawing code may need to be aware of occurs.

When your `DirectConnected()` function is called, it's passed a pointer to a `direct_buffer_info` structure, as follows:

```
typedef struct {
```

```
   direct_buffer_state   buffer_state;
   direct_driver_state   driver_state;
   void                  *bits;
   void                *pci_bits;
   int32                   bytes_per_row;
   uint32                bits_per_pixel;
   color_space               pixel_format;
   buffer_layout               layout;
   buffer_orientation   orientation;
   uint32                  _reserved[9];
   uint32                _dd_type_;
   uint32                _dd_token_;
   uint32                clip_list_count;
   clipping_rect               window_bounds;
   clipping_rect               clip_bounds;
   clipping_rect               clip_list[1];
} direct_buffer_info;
```

buffer_state indicates what change is occurring in the direct buffer access privileges. It can have one of the following values:

You will always receive a **B_DIRECT_START** notification when your BDirectWindow is first connected to the screen, followed by any number of **B_DIRECT_MODIFY** notifications (it's possible you won't receive any at all). When you return from **DirectConnected()** after handling **B_DIRECT_START** or **B_DIRECT_MODIFY**, your application guarantees to the application server that your code will abide by the frame buffer configuration specified by the **direct_buffer_info** structure until another **B_DIRECT_MODIFY** notification is received (or **B_DIRECT_STOP** occurs).

You'll receive a **B_DIRECT_STOP** notification when the window is closed, hidden, or if moved, or if a resolution or color depth change occurs. Once your **DirectConnected()** function returns from handling this notification, you guarantee to the application server that your code won't touch the frame buffer anymore.

For any of these notifications, your **DirectConnected()** function shouldn't return until you can guarantee to the application server that your code will abide by the frame buffer configuration it received.

You can get further information about what changed by testing against other flags in the **buffer_state** field:

Constant	Description
B_BUFFER_MOVED	The content of your window has been moved, either by a call to **MoveTo()** or **MoveBy()** or by the user manually dragging the window. The contents of the window are always moved relative to the top-left corner of the window.
B_BUFFER_RESET	The entire direct access buffer has been reset. This can happen if the user changes the depth or resolution of the screen, or if the window had previously been hidden and has been made visible again.

B_BUFFER_RESIZED	The content area of your window has been resized.
B_CLIPPING_MODIFIED	The visible region of the content area of your window changed. This doesn't imply anything about the position of the window or the size of the content area of the window—it simply means that the part of the window that's visible has changed shape.

The **driver_state** field indicates changes in the state of the graphics card on which your direct window is displayed. There are two possible values:

Constant	Description
B_MODE_CHANGED	The resolution or depth of the graphics card has changed.
B_DRIVER_CHANGED	The window was moved onto another monitor.

The **bits** field is a pointer to the frame buffer in your own team's memory space.

The **pci_bits** field is a pointer to the frame buffer in the PCI memory space; this value is typically needed to control DMA.

bytes_per_row is the number of bytes used to represent a single row of pixels in the frame buffer.

bits_per_pixel is the number of bits actually used to store a single pixel, including reserved, unused, or alpha channel bits. This value is usually a multiple of eight.

pixel_format is the format used to encode a pixel, as defined in the **color_space** type in <**GraphicsDefs.h**>.

The **layout** , **orientation** , **_reserved** , **_dd_type_** , and **_dd_token_** fields are all reserved for future use and must not be used.

window_bounds is a rectangle that defines the full content area of the window, in screen coordinates. You can convert these coordinates into frame buffer addresses using the values in the **bits** , **bytes_per_row** , and **bits_per_pixel** fields. The **clipping_rect** structure is:

```
typedef struct {
   int32      left;
   int32      top;
   int32      right;
   int32      bottom;
} clipping_rect;
```

Note that, as always, these edges are inclusive; for example, if left is 5 and top is 3, the pixel at (5,3) is included in the rectangle's contents.

clip_bounds is the bounding rectangle of the visible part of the content area of the window, in screen coordinates. This rectangle is the smallest rectangle that contains all the rectangles in the **clip_list** , described below.

clip_list_count is the number of rectangles in the **clip_list** . The **clip_list** is a list of rectangles that together define the visible region of the content area of the window, in screen coordinates

The data in the **direct_buffer_info** structure is only valid until **DirectConnected()** returns, so if you need to reference any of the information later, you should make a copy of the fields you need.

If your **DirectConnected()** implementation doesn't handle a request within three seconds, the Application Server will intentionally crash your application under the assumption that it's deadlocked. Be sure to handle requests as quickly as possible.

Window Mode vs. Full Screen Mode

There are some differences in how BDirectWindow behaves depending on whether it's in window mode or full-screen exclusive mode.

In window mode, the BDirectWindow behaves almost exactly like a BWindow—so much so that you can use a BDirectWindow in any situation you'd normally use a BWindow. The **window_bounds** rectangles are the same size and shape as the window itself, as you'd expect. If exclusive window mode is available (**SupportsWindowMode()** returns true), **DirectConnected()** will be called as described above, thereby providing the means to directly access the frame buffer. If the graphics card doesn't support exclusive access to the frame buffer while in window mode, **DirectConnected()** will never be called, and you can only use BWindow and BView APIs to work in the window.

In full-screen exclusive mode, the **window_bounds** are actually the size and shape of the entire screen, even if the screen isn't the same size as the direct window you created. You have to handle the difference yourself.

Full-screen exclusive mode also guarantees that your window will always be the focus, always be in front, and will always stay full-screen (the application server will resize the window for you if the screen resolution changes). Since no other windows can come in front of a full-screen exclusive direct window, any Interface Kit objects that use a window to display their contents won't work; this includes any type of menu.

If you want your BDirectWindow to be full-screen, but still compatible with menus or other windows, create it as a non-exclusive window, then use the following code:

```
BScreen screen(this);
MoveTo(0,0);
ResizeTo(screen.width, screen.height);
```

This will make the non-exclusive direct window fill the entire screen. Keep in mind that in this case, other windows may appear in front of yours, and if the screen resolution changes, you will have to resize the window yourself if you want to continue to fill the entire screen.

Using a Direct Window

Let's put together a simple class, derived from BDirectWindow, that demonstrates the basics of drawing into a direct window.

```
class DirectSample : public BDirectWindow {
   public:
                DirectSample(BRect frame);
                ~DirectSample();
      virtual bool   QuitRequested();
      virtual void   DirectConnected(direct_buffer_info *info);

      uint8         *fBits;
      int32         fRowBytes;
      color_space      fFormat;
      clipping_rect   fBounds;

      uint32         fNumClipRects;
      clipping_rect   *fClipList;

      bool         fDirty;      // needs refresh?
      bool         fConnected;
      bool         fConnectionDisabled;
      BLocker         *locker;
      thread_id      fDrawThreadID;
};
```

The DirectSample class implements a constructor as well as the **QuitRequested()** and **DirectConnected()** functions.

Some variables are added to the class for cacheing information about the frame buffer.

- **fBits** will contain a pointer to the frame buffer's bitmap.
- **fRowBytes** will contain the number of bytes per row of screen data.
- **fFormat** will contain the pixel format (such as **B_CMAP8** for 8-bit indexed color graphics mode). Our sample program will only work in this mode.
- **fBounds** will contain the bounds rectangle for the window.
- **fNumClipRects** will contain the number of rectangles in the clip rectangle list.
- **fClipList** is the actual list of clip rectangles, and will be allocated on-the-fly as needed.
- **fDirty** will be **true** if the window needs to be redrawn.
- **fConnected** is **true** if the window is connected to the frame buffer.
- **fConnectionDisabled** is **true** if the window is in the process of being closed.

- **`locker`** is a BLocker that will be used to ensure mutual exclusion when the frame buffer or buffer information data we've cached is being manipulated.
- **`fDrawThreadID`** contains the **`thread_id`** of the drawing thread, which is responsible for drawing the contents of the window.

The specifics of what these variables are and why the information contained in them is maintained will be discussed when we get to the **`DirectConnected()`** and `DrawingThread()` functions.

The constructor for the DirectSample class looks like this:

```
DirectSample::DirectSample(BRect frame)
         : BDirectWindow(frame, "DirectWindow Sample",
               B_TITLED_WINDOW,
                     B_NOT_RESIZABLE|B_NOT_ZOOMABLE) {

   fConnected = false;
   fConnectionDisabled = false;
   locker = new BLocker();
   fClipList = NULL;
   fNumClipRects = 0;

   AddChild(new SampleView(Bounds()));

   if (!SupportsWindowMode()) {
      SetFullScreen(true);
   }

   fDirty = true;
   fDrawThreadID = spawn_thread(DrawingThread, "drawing_thread",
                  B_NORMAL_PRIORITY, (void *) this);
   resume_thread(fDrawThreadID);
   Show();
}
```

This code establishes the direct window by deferring to BDirectWindow. Then the `fConnected` and `fConnectionDisabled` flags are initialized to indicate that the window isn't connected yet, but the connection isn't in the process of being torn down by the DirectSample destructor. The `locker` is created, and the clip rectangle list is initialized to a **`NULL`** pointer, with a count of 0.

Then it adds a child view that occupies the entire window. The primary purpose of this view in this sample is to set the view color to **`B_TRANSPARENT_32_BIT`**, to prevent the application server from erasing the window with a default color.

If the video card doesn't support window mode, we call **`SetFullScreen()`** to switch the direct window into full-screen exclusive mode. This guarantees that you'll get connected with direct screen access (in a window if possible, otherwise in full-screen exclusive mode). If you don't

use SetFullScreen(), and window mode isn't supported, DirectConnected() will never be called, and you won't have direct screen access.

Then the **fDirty** flag is set to true, which indicates that the window needs to be updated, and the drawing thread is started; the drawing thread will handle all actual drawing into the window. The argument passed to the drawing thread is a pointer to the DirectSample window itself. You should always use a separate thread for drawing into a BDirectWindow.

Finally, Show() is called to make the direct window visible.

The destructor needs to make sure there's no chance someone will try to draw while the window is being destructed:

```
DirectSample::~DirectSample() {
   int32 result;

   fConnectionDisabled = true;      // Connection is dying
   Hide();
   Sync();
   wait_for_thread(fDrawThreadID, &result);
   free(fClipList);
   delete locker;
}
```

The first thing the destructor does is set the **fConnectionDisabled** flag to **true**, which indicates that the window is in the process of being destroyed, and that future calls to DirectConnected() or the drawing thread should be ignored. The window is then hidden by calling Hide(). Finally, Sync() is called to block until the window is actually hidden.

wait_for_thread() waits until the drawing thread terminates. The drawing thread (as we'll see shortly) is designed to terminate when the `fConnectionDisabled` flag is **true**.

Then the clip rectangle list is freed and the locker deleted.

The QuitRequested() function is implemented as usual.

The DirectConnected() function is called whenever a change occurs that affects how your code should access the frame buffer:

```
void DirectSample::DirectConnected(direct_buffer_info *info) {
   if (!fConnected && fConnectionDisabled) {
      return;
   }
   locker->Lock();

   switch(info->buffer_state & B_DIRECT_MODE_MASK) {
      case B_DIRECT_START:
         fConnected = true;
      case B_DIRECT_MODIFY:
```

```
        // Get clipping information

        if (fClipList) {
           free(fClipList);
           fClipList = NULL;
        }
        fNumClipRects = info->clip_list_count;
        fClipList = (clipping_rect *)
              malloc(fNumClipRects*sizeof(clipping_rect));
        if (fClipList) {
           memcpy(fClipList, info->clip_list,
                 fNumClipRects*sizeof(clipping_rect));
           fBits = (uint8 *) info->bits;
           fRowBytes = info->bytes_per_row;
           fFormat = info->pixel_format;
           fBounds = info->window_bounds;
           fDirty = true;

        }
        break;
     case B_DIRECT_STOP:
        fConnected = false;
        break;
   }
   locker->Unlock();
}
```

DirectConnected() begins by checking the `fConnected` and `fConnectionDisabled` flags; the code in DirectConnected() is only run if the connection is opened (`fConnected` is **`true`**) or if we want to start it again (`fConnectionDisabled` is **`false`**). This arrangement prevents the DirectConnected() function from trying to reconnect if the destructor has started running. Otherwise, the locker is locked, to prevent DirectConnected() and the drawing thread from colliding.

If the buffer state is **`B_DIRECT_START`**, the `fConnected` flag is set to **`true`**. This keeps track of the fact that the application server has given permission to draw directly into the region of the frame buffer controlled by the direct window.

If the buffer state is **`B_DIRECT_START`** or **`B_DIRECT_MODIFY`** (in which case the direct_buffer_info structure describes changes to the frame buffer), any previously-existing clip rectangle list is deleted, then we cache the information that interests us and set the **`fDirty`** flag to **`true`** (to indicate that the display needs to be redrawn to reflect the changed graphics settings).

The clip list is also cached by saving the number of rectangles in the list in the `fNumClipRects` field and by making a copy of the clip list into a newly malloc()d block of memory.

If the state is **`B_DIRECT_STOP`**, the **`fConnected`** flag is set to **`false`**, to indicate that we shouldn't draw into the frame buffer anymore.

Finally, the locker is unlocked, which lets the drawing thread start running again.

Now let's have a look at `DrawingThread()`; this function serves as the drawing thread, and is a global function:

```
int32 DrawingThread(void *data) {
   DirectSample *w;

   w = (DirectSample *) data;
   while (!w->fConnectionDisabled) {
      w->locker->Lock();
      if (w->fConnected) {
         if (w->fFormat == B_CMAP8 && w->fDirty) {
            int32 y;
            int32 width;
            int32 height;
            int32 adder;
            uint8 *p;
            clipping_rect *clip;
            int32 i;

            adder = w->fRowBytes;   // Stash locally for this pass
            for (i=0; i<w->fNumClipRects; i++) {
               clip = &(w->fClipList[i]);
               width = (clip->right - clip->left)+1;
               height = (clip->bottom - clip->top)+1;
               p = w->fBits+(clip->top*w->fRowBytes)+clip->left;
               y = 0;
               while (y < height) {
                  memset(p, 0x00, width);
                  y++;
                  p+=adder;
               }
            }
         }
         w->fDirty = false;
      }
      w->locker->Unlock();
      // Use BWindow or BView APIs here if you want
      snooze(16000);
   }
   return B_OK;
}
```

`DrawingThread()` starts by casting the argument, *data*, into a pointer to the DirectSample window into which it will be drawing.

The while loop that follows will continue to run as long as the `fConnectionDisabled` flag is **`true`**—in other words, it will keep looping as long as the connection is enabled.

The drawing loop itself begins by locking the locker to ensure that **DirectConnected()** doesn't change anything while we're working, then checking to be sure the connection is opened (`fConnected` is **true**). If the connection is open, we verify that format of the window is still 8-bit color and that the display needs to be updated. If the display needs updating and the pixel format is still **B_CMAP8**, the drawing code begins.

The `fRowBytes` field of the DirectSample window is cached in a local variable called `adder`. Then each rectangle in the clip list is drawn, one at a time, using a for loop.

A pointer to the clip rectangle to be drawn is stored in `clip`, and the `width` and `height` of the rectangle are computed. Then `p` is set to be a pointer to the first pixel in the frame buffer that's contained by the clip rectangle. Since 8-bit color pixels each occupy exactly one byte of video memory, this pixel's address can be computed by taking the base `fBits` pointer, adding the number of bytes per row times the number of rows between the top of the screen and the first row in the clip rectangle, then adding the number of bytes between the left edge of the screen and the left edge of the clip rectangle, as seen in the line:

```
p = w->fBits+(clip->top*w->fRowBytes)+clip->left;
```

Then a while loop is used to iterate over each line in the clip rectangle, by ranging the variable `y` from `0` to the `height` of the clip rectangle. **memset()** is used to clear the row to black, which is represented by the byte value 0x00. `y` is incremented by one for each pass through the loop, to count the rows being drawn for each iteration, and the pointer `p` is incremented by `adder` to move down to the beginning of the next row in the clip rectangle.

Once each clip rectangle has been drawn, the for loop exits, and the `fDirty` flag is set to false to indicate that the screen is up-to-date. Once that's done, the locker is unlocked, which lets **DirectConnected()** do its thing if it's called. To avoid using an unreasonable amount of processing time, **snooze()** is called to give up CPU time to other threads.

If you want to use calls to BWindow or BView API in your drawing thread, you should do so just after unlocking the window.

When the thread terminates (which will only happen when the connection is disabled), the drawing thread returns **B_OK**.

This drawing function is designed to draw nothing unless it's necessary; **DirectConnected()** will set the `fDirty` flag when something happens to cause the screen to need a refresh, and other code elsewhere in the application could also set the fDirty flag to indicate that the screen should be redrawn.

Since we're taking over drawing the contents of the window, we need to tell the application server not to draw anything. This is done by adding the following line to the constructor for the SampleView:

```
SetViewColor(B_TRANSPARENT_32_BIT);
```

This is very important: if you don't remember to do this, you'll have all kinds of synchronization problems when the application server and your drawing code try to draw into

the window at the same time.

Note that this sample code doesn't really do anything useful (if all you want to do is have a black window moving around, don't use BDirectWindow—use a regular BWindow, throw a BView into it, and use **`SetViewColor()`** to make the view black; it'll be faster and more efficient because it will use hardware graphics acceleration if it's available). However, it serves as a simple example of how to establish a connection to let your own drawing code directly access the screen. Just replace the code inside the drawing loop with something more useful (like a nifty real-time animation).

Hook Functions

`DirectConnected()`

The **`DirectConnected()`** hook function is called when the connection to the screen has been made, a change has occurred to the size or format of the frame buffer, a change has occurred to the position, size, or shape of the visible part of the content area of the window, or the connection to the screen is terminated.

Constructor and Destructor

BDirectWindow()

```
BDirectWindow(BRect frame, const char *title, window_type type, uint32 flags,
    uint32 workspace = B_CURRENT_WORKSPACE)
BDirectWindow(BRect frame, const char *title, window_look look,
    window_feel feel, uint32 flags,
    uint32 workspace = B_CURRENT_WORKSPACE)
```

Creates and returns a new BDirectWindow object. This is functionally equivalent to the BWindow constructor, except the resulting BDirectWindow supports direct window operations.

You will probably want to set up a flag to keep track of whether or not the direct window's connection to the screen is viable. In the constructor, you should set this flag (let's call it **fConnectionDisabled**) to **false**, which indicates to both **DirectConnected()** and your drawing thread that the window is not in the process of being deconstructed. The destructor would then set this flag to **true** before terminating the connection to avoid the unlikely possibility of the connection trying to restart while the BDirectWindow is being dismantled.

You'll also need other flags or semaphores (or benaphores) to manage the interaction between the BDirectWindow and your drawing thread.

See the sample code in "Using a Direct Window" on page9 for an example.

~BDirectWindow()

```
~BDirectWindow()
```

Frees all memory the BDirectWindow object allocated for itself. You should never `delete` a BDirectWindow object; call its **Quit()** or **Close()** function instead.

Your BDirectWindow destructor should begin by setting the **fConnectionDisabled** flag to **true**, to prevent **DirectConnected()** from attempting to reconnect to the direct window while it's being deconstructed.

Then you should call **Hide()** and **Sync()** to force the direct window to disconnect direct access:

```
MyDirectWindow::~BDirectWindow {
   fConnectionDisabled = true;
   Hide();
   Sync();
   /* complete usual destruction here */
}
```

Member Functions

DirectConnected()

virtual void **DirectConnected**(direct_buffer_info **info*)

This hook function is the core of BDirectWindow. Your application should override this function to learn about the state of the graphics display onto which you're drawing, as well as to be informed of any changes that occur.

This function is also called to suspend and resume your direct access privileges.

Your code in this function should be as short as possible, because what your `DirectConnected()` function does can affect the performance of the entire system. `DirectConnected()` should only handle the immediate task of dealing with changes in the direct drawing context, and shouldn't normally do any actual drawing—that's what your drawing thread is for.

If you have drawing that absolutely has to be done before you can safely return control to the application server (see the note below), you may do so, but your code should do the absolute minimum drawing necessary and leave everything else to the drawing thread.

`DirectConnected()` should only return when it can guarantee to the application server that the request specified by *info* will be strictly obeyed.

The structure pointed to by *info* goes away after `DirectConnected()` returns, so you should cache the information that interests you.

If your **`DirectConnected()`** implementation doesn't handle a request within three seconds, the Application Server will intentionally crash your application under the assumption that it's deadlocked. Be sure to handle requests as quickly as possible.

See "Getting Connected (and Staying That Way)" on page5 for more information about the **`direct_buffer_info`** structure.

GetClippingRegion()

status_t **GetClippingRegion**(BRegion **region*, BPoint **origin* = NULL) const

Sets the specified *region* to match the current clipping region of the direct window. If *origin* is specified, each point in the region is offset by the *origin*, resulting in a BRegion that's localized to your application's vision of where in space the origin is (relative to the origin of the screen's frame buffer).

Although the **direct_buffer_info** structure contains the clipping region of a direct window, it's not in standard BRegion form. This function is provided so you can obtain a standard BRegion if you need one.

The `GetClippingRegion()` function can only be called from the **`DirectConnected()`** function; calling it from outside **`DirectConnected()`** will return invalid results.

If you need to cache the clipping region of your window and need a BRegion for clipping purposes, you could use the following code inside your **`DirectConnected()`** function:

```
BRegion rgn;
GetClippingRegion(&rgn);
```

This serves a double purpose: it obtains the clipping region in BRegion form, and it returns a copy of the region that you can maintain locally. However, it may be more efficient to copy the clipping region by hand, since the clipping rectangle list used by BDirectWindow uses integer numbers, while BRegion uses floating-point.

RETURN CODES

- **`B_OK`**. The clipping region was successfully returned.
- **`B_ERROR`**. An error occurred while trying to obtain the clipping region.

IsFullScreen() , SetFullScreen()

bool **`IsFullScreen`**(void) const
status_t **`SetFullScreen(`**bool *enable*`)`

`IsFullScreen()` returns **`true`** if the direct window is in full-screen exclusive mode, or **`false`** if it's in window mode.

The value returned by `IsFullScreen()` is indeterminate if a call to `SetFullScreen()` is in progress—if this is the case, you shouldn't rely on the resulting value. Instead, it would be safer to maintain a state setting of your own and use that value.

`SetFullScreen()` enables full-screen exclusive mode if the *enable* flag is **`true`**. To switch to window mode, pass **`false`**. The **`SupportsWindowMode()`** function can be used to determine whether or not the video card is capable of supporting window mode. See "Window Mode vs. Full Screen Mode" on page8 for a detailed explanation of the differences between these modes.

When your window is in full screen mode, it will always have the focus, and no other window can come in front of it.

`SetFullScreen()` can return any of the following result codes.

RETURN CODES

- B_OK. The mode was successfully changed.
- B_ERROR. An error occurred while trying to switch between full screen and window modes (for example, another window may already be in full-screen exclusive mode in the same workspace).

SetFullScreen() see IsFullScreen()

SupportsWindowMode()

```
static bool SupportsWindowMode(screen_id id = B_MAIN_SCREEN_ID)
```

Returns **true** if the specified screen supports window mode; if you require the ability to directly access the frame buffer of a window (rather than occupying the whole screen), you should call this function to be sure that the graphics hardware in the computer running your application supports it. Because this is a static function, you don't have to construct a BDirectWindow object to call it:

```
if (BDirectWindow::SupportsWindowMode()) {
   /* do stuff here */
}
```

In particular, window mode requires a graphics card with DMA support and a hardware cursor; older video cards may not be capable of supporting window mode.

If window mode isn't supported, but you still select window mode, DirectConnected() will never be called (so you'll never be authorized for direct frame buffer access).

Even if window mode isn't supported, you can still use BDirectWindow objects for full-screen direct access to the frame buffer, but it's recommended that you avoid direct video DMA or the use of parallel drawing threads that use both direct frame buffer access and BView calls (because it's likely that such a graphics card won't handle the parallel access and freeze the PCI bus—and that would be bad).

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Game Kit Table of Contents

- The Game Kit
- BDirectWindow
- BFileGameSound
- BGameSound
- BPushGameSound
- BSimpleGameSound
- BStreamingGameSound
- BWindowScreen
- Global Functions, Constants, and Defined Types
- Graphics Card Hook Functions

The Be Book **The Game Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Game Kit: Master Index

_

_dd_token_	BDirectWindow
_dd_type_	BDirectWindow
_reserved	BDirectWindow

B

bits	BDirectWindow
bits_per_pixel	BDirectWindow
blit()	Graphics Card Hook Functions
buffer_size	Global Functions, Constants, and Defined Types
buffer_state	BDirectWindow
byte_order	Global Functions, Constants, and Defined Types
bytes_per_row	BDirectWindow

C

CanControlFrameBuffer()	BWindowScreen
CardHookAt()	BWindowScreen
CardInfo()	BWindowScreen
channel_count	Global Functions, Constants, and Defined Types

Choosing a Player Class	BGameSound
clip_bounds	BDirectWindow
clip_list	BDirectWindow
clip_list_count	BDirectWindow
Clone()	BGameSound
ColorList()	BWindowScreen
Constants	BPushGameSound
Constants	Global Functions, Constants, and Defined Types
Constructor and Destructor	BDirectWindow
Constructor and Destructor	BFileGameSound
Constructor and Destructor	BGameSound
Constructor and Destructor	BPushGameSound
Constructor and Destructor	BSimpleGameSound
Constructor and Destructor	BStreamingGameSound
Constructor and Destructor	BWindowScreen
Control Flags	Global Functions, Constants, and Defined Types
Coordinate Spaces	Graphics Card Hook Functions
B_CRT_CONTROL	Global Functions, Constants, and Defined Types

CurrentPosition()	BPushGameSound

D

Debugging	BWindowScreen
define_cursor()	Graphics Card Hook Functions
Defined Types	Global Functions, Constants, and Defined Types
Device()	BGameSound
DirectConnected()	BDirectWindow
BDirectWindow	BDirectWindow
BDirectWindow()	BDirectWindow
~BDirectWindow()	BDirectWindow
direct_buffer_info	BDirectWindow
Disconnect()	BWindowScreen
draw_array_with_32_bit_depth()	Graphics Card Hook Functions
draw_array_with_8_bit_depth()	Graphics Card Hook Functions
draw_line_with_16_bit_depth()	Graphics Card Hook Functions
draw_line_with_32_bit_depth()	Graphics Card Hook Functions
draw_line_with_8_bit_depth()	Graphics Card Hook Functions
draw_rect_with_16_bit_depth()	Graphics Card Hook Functions
draw_rect_with_32_bit_depth()	Graphics Card Hook Functions
draw_rect_with_8_bit_depth()	Graphics Card Hook Functions
driver_state	BDirectWindow

E

Exclusive Access	BPushGameSound

F

fConnectionDisabled	BDirectWindow
BFileGameSound	BFileGameSound
BFileGameSound()	BFileGameSound
~BFileGameSound	BFileGameSound
FillBuffer()	BStreamingGameSound
Format()	BGameSound
format	Global Functions, Constants, and Defined Types
FrameBufferInfo()	BWindowScreen
B_FRAME_BUFFER_CONTROL	Global Functions, Constants, and Defined Types
frame_buffer_info	Global Functions, Constants, and Defined Types
frame_rate	Global Functions, Constants, and Defined Types

G

Gain()	BGameSound
The Game Kit	The Game Kit
The Game Kit	The Game Kit
BGameSound	BGameSound
BGameSound()	BGameSound

~BGameSound	BGameSound
B_GAMMA_CONTROL	Global Functions, Constants, and Defined Types
GetAttributes()	BGameSound
GetClippingRegion()	BDirectWindow
Getting Connected (and Staying That Way)	BDirectWindow
Getting Connected (and Staying That Way)	BDirectWindow
Global Functions	Global Functions, Constants, and Defined Types
Global Functions, Constants, and Defined Types	Global Functions, Constants, and Defined Types
Global Functions, Constants, and Defined Types	Global Functions, Constants, and Defined Types
Graphics Card Hook Functions	Graphics Card Hook Functions
Graphics Card Hook Functions	Graphics Card Hook Functions
graphics_card_hook	Global Functions, Constants, and Defined Types
The graphics_card_hook Type	Graphics Card Hook Functions
graphics_card_info	Global Functions, Constants, and Defined Types
gs_attribute	Global Functions, Constants, and Defined Types
gs_attribute_info	Global Functions, Constants, and Defined Types
gs_attributes	Global Functions, Constants, and Defined Types

gs_audio_format	Global Functions, Constants, and Defined Types
B_GS_CD_THROUGH_GAIN	Global Functions, Constants, and Defined Types
B_GS_F	Global Functions, Constants, and Defined Types
B_GS_FIRST_PRIVATE_ATTRIBUTE	Global Functions, Constants, and Defined Types
B_GS_FIRST_USER_ATTRIBUTE	Global Functions, Constants, and Defined Types
B_GS_GAIN	Global Functions, Constants, and Defined Types
gs_id	Global Functions, Constants, and Defined Types
B_GS_LOOPING	Global Functions, Constants, and Defined Types
B_GS_MAIN_GAIN	Global Functions, Constants, and Defined Types
B_GS_PAN	Global Functions, Constants, and Defined Types
B_GS_S16	Global Functions, Constants, and Defined Types
B_GS_S32	Global Functions, Constants, and Defined Types
B_GS_SAMPLING_RATE	Global Functions, Constants, and Defined Types
B_GS_U8	Global Functions, Constants, and Defined Types

H

Hide()	BWindowScreen
High-Performance Audio	The Game Kit
Hook Functions	BDirectWindow
Hook Functions	BWindowScreen
Hook Functions	Graphics Card Hook Functions
How It Works	BPushGameSound

I

ID()	BGameSound
IOBase()	BWindowScreen
indexed_color_line	Graphics Card Hook Functions
Init()	BGameSound
InitCheck()	BFileGameSound
InitCheck()	BGameSound
InitCheck()	BPushGameSound
InitCheck()	BSimpleGameSound
InitCheck()	BStreamingGameSound
invert_rect()	Graphics Card Hook Functions
IsFullScreen()	BDirectWindow
IsLooping()	BSimpleGameSound
IsPaused()	BFileGameSound
IsPlaying()	BGameSound

K

Keyboard and Mouse	BWindowScreen

L

layout	BDirectWindow
Lock()	BStreamingGameSound
LockForCyclic()	BPushGameSound
LockMemoryPool()	BGameSound
LockNextPage()	BPushGameSound
lock_status	BPushGameSound
Low-Level Graphics Access	The Game Kit

M

Member Functions	BDirectWindow
Member Functions	BFileGameSound
Member Functions	BGameSound
Member Functions	BPushGameSound
Member Functions	BSimpleGameSound
Member Functions	BStreamingGameSound
Member Functions	BWindowScreen
MoveDisplayArea()	BWindowScreen
move_cursor()	Graphics Card Hook Functions

N

Naming	Graphics Card Hook Functions
The Neverending Story	BPushGameSound

O

orientation	BDirectWindow

P

Pan()	BGameSound
pci_bits	BDirectWindow
Performance Verification	BGameSound
pixel_format	BDirectWindow
Polyphony	BGameSound
Preload()	BFileGameSound
BPushGameSound	BPushGameSound
BPushGameSound()	BPushGameSound
~BPushGameSound	BPushGameSound

Q

Quit()	BWindowScreen

R

RegisterThread()	BWindowScreen
rgb_color_line	Graphics Card Hook Functions

S

Screen Access	BWindowScreen
ScreenChanged()	BWindowScreen
ScreenConnected()	BWindowScreen
SetAttributes()	BGameSound
SetAttributes()	BStreamingGameSound
SetColorList()	BWindowScreen
SetFrameBuffer()	BWindowScreen
SetFullScreen()	BDirectWindow
SetGain()	BGameSound
SetInitError()	BGameSound
SetIsLooping()	BSimpleGameSound
SetMaxSoundCount()	BGameSound
SetMemoryPoolSize()	BGameSound
SetPan()	BGameSound
SetParameters()	BStreamingGameSound
SetPaused()	BFileGameSound
SetSpace()	BWindowScreen
SetStreamHook()	BStreamingGameSound
set_mouse_position()	Global Functions, Constants, and Defined Types
Show()	BWindowScreen

show_cursor()	Graphics Card Hook Functions
BSimpleGameSound	BSimpleGameSound
BSimpleGameSound()	BSimpleGameSound
~BSimpleGameSound	BSimpleGameSound
The Sound of Explosions in the Morning	BGameSound
StartPlaying()	BGameSound
StopPlaying()	BGameSound
BStreamingGameSound	BStreamingGameSound
BStreamingGameSound()	BStreamingGameSound
~BStreamingGameSound	BStreamingGameSound
SupportsWindowMode()	BDirectWindow
Suspend()	BWindowScreen
SuspensionHook()	BWindowScreen
sync()	Graphics Card Hook Functions

U

Unimplemented Functions	Graphics Card Hook Functions
Unlock()	BStreamingGameSound
UnlockCyclic()	BPushGameSound
UnlockPage()	BPushGameSound
Using a Direct Window	BDirectWindow
Using a Direct Window	BDirectWindow

W

WindowActivated()	BWindowScreen
Window Mode vs. Full Screen Mode	BDirectWindow
Window Mode vs. Full Screen Mode	BDirectWindow
BWindowScreen	BWindowScreen
BWindowScreen()	BWindowScreen
~BWindowScreen()	BWindowScreen
window_bounds	BDirectWindow
WorkspaceActivated()	BWindowScreen
Workspaces	BWindowScreen

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Graphics Card Hook Functions

Declared in: be/add-ons/graphics/GraphicsCard.h
more...

A graphics card driver can implement a set of hook functions that perform specific tasks on behalf of the Application Server and the Game Kit's BWindowScreen object. Drivers should implement as many of these functions as they can.

The Application Server asks the driver for the locations of its hook functions by passing the **`B_GET_GRAPHICS_CARD_HOOKS`** opcode. In response to this request, the driver writes an array of function pointers to *data*, cast as a **`graphics_card_hook`** pointer (described below). The **`B_GET_GRAPHICS_CARD_HOOKS`** request is made whenever the configuration of the frame buffer changes. The driver can thus provide hook functions that are tailored to specific frame buffer configurations.

The system can accommodate **`B_GRAPHICS_HOOK_COUNT`** (48) hook functions. Currently, only the first 14 functions are used. These functions fall into four groups:

- **Indices 0–2: Cursor management**. Drivers must implement all three of these functions, or none of them.
- **Indices 3–9, 12, and 13: Drawing functions**. These are specific drawing tasks, such as drawing a line or filling a rectangle. A driver can implement as many of these as it wishes.
- **Index 10: Driver/Application Server synchronization**. Drivers should implement this function only if the other hook functions are performed asynchronously.
- **Index 11: Color inversion**. This function inverts the colors in a rectangle.

The graphics_card_hook Type

All pointers in the hook function array are declared to be of type **graphics_card_hook**:

```
typedef void (*graphics_card_hook)(void)
```

The code that fills *data* will look something like this:

```
int32 control_graphics_card(uint32 opcode, void *data)
{
   switch (opcode) {
   ...
   case B_GET_GRAPHICS_CARD_HOOKS:
      ((graphics_card_hook *)data)[0] = (graphics_card_hook)define_cursor;
      ((graphics_card_hook *)data)[1] = (graphics_card_hook)move_cursor;
   ...
   }
}
```

Despite the **graphics_card_hook** declaration, each function has its own set of arguments and returns an integer error code. All functions should return **B_OK** if they're successful, and **B_ERROR** if not.

Unimplemented Functions

All hook functions that the driver doesn't implement (including indices 14–47) should be set to **`NULL`**. For example:

```
((graphics_card_hook *)data)[14] = (graphics_card_hook)NULL;
```

A driver can chose to nullify functions it *does* implement if it wants to defer to the Application Server version.

Don't implement a function to simply (always) return **`B_ERROR`**. If you want to declare a hook as a no-op, you must pass **`NULL`** as the pointer.

Coordinate Spaces

Most hook function coordinates are in depth-independent "frame buffer space" (the exceptions are well noted below). In other words, a coordinate pair gives the location of a pixel in the frame buffer independent of the buffer's depth; pixel (0, 0) is at the left top corner of the frame buffer.

Naming

You can name the functions whatever you wish—the Application Server finds the functions by index in the hook function array. For convenience and clarity, the following descriptions suggest default names.

Hook Functions

define_cursor()

Index: 0

```
int32 define_cursor(uchar *xorMask, uchar *andMask,
    int32 width, int32 height,
    int32 hotX, int32 hotY)
```

Tells the driver to create a cursor image as defined by the arguments. The first two arguments, *xorMask* and *andMask*, are bit vectors that represent the cursor image laid out in concatenated, byte-aligned rows (top to bottom). Parallel bits from the two vectors define the color of a single cursor pixel:

xorMask	*andMask*	Meaning
0	0	White; replace the screen pixel with a white cursor pixel.
1	0	Black; replace the screen pixel with a black cursor pixel.
0	1	Transparent; let the color of the underlying screen pixel show through.
1	1	Inversion; invert the color of the screen pixel.

The second two arguments, *width* and *height*, are the size of the cursor image in pixels. (Currently, the Application Server supports only 16x16 cursors.)

The (*hotX*, *hotY*) arguments define the "hot spot"—the pixel that precisely locates the cursor. Hot spot coordinates are relative to the cursor rectangle itself, where the pixel at the left top corner of the cursor image is (0, 0) and the one at the right bottom corner is (*width*-1, *height*-1).

If the cursor is currently showing (i.e. not hidden), this function should display the cursor image.

move_cursor()

Index: 1

```
int32 move_cursor(int32 screenX, int32 screenY)
```

Tells the driver to move the cursor so the hot spot corresponds to (*screenX*, *screenY*). The arguments are display area coordinates (*not* frame buffer coordinates).

show_cursor()

Index: 2

```
int32 show_cursor(bool flag)
```

If the *flag* argument is **true**, the driver should show the cursor image on-screen; if it's **false**, it should remove the cursor from the screen.

If the driver is asked to show the cursor before **define_cursor()** is called, it should show it at (0, 0).

---

draw_line_with_8_bit_depth()

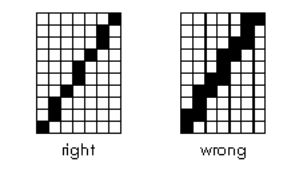
Index: 3

```
int32 draw_line_with_8_bit_depth(int32 startX, int32 endX,
    int32 startY, int32 endY,
    uint8 colorIndex,
    bool clipToRect,
    int16 clipLeft, int16 clipTop, int16 clipRight, int16 clipBottom)
```

Tells the driver to draw a straight, 8-bit color, minimally thin line.

- The line begins at (*startX*, *startY*) and ends at (*endX*, *endY*), inclusive. The arguments are frame buffer coordinates.
- *colorIndex* gives the color of the line as an index into the 8-bit color table.
- If *clipToRect* is **true**, the function should draw only the portion of the line that lies within the clipping rectangle defined by the last four arguments. The sides of the rectangle are included in the drawing area. If *clipToRect* is **false**, the final four arguments should be ignored

To produce minimal thinness, the line should color only one pixel per row or column, as the absolute slope of the line is more or less than 45 degrees; in other words, the line should move between rows or columns on the diagonal, not by overlapping. Here's how you should (and shouldn't) produce a mostly-vertical line; for the mostly-horizontal version, turn your head sideways:



---

draw_line_with_32_bit_depth()

Index: 4

```
int32 draw_line_with_32_bit_depth(int32 startX, int32 endX,
    int32 startY, int32 endY,
    uint32 color,
    bool clipToRect,
    int16 clipLeft, int16 clipTop, int16 clipRight, int16 clipBottom)
```

This is the same as **draw_line_with_8_bit_depth()** except for the *color* argument. Here, *color* is a 32-bit value with 8-bit red, green, blue, and alpha components. The components are arranged in the order that the driver specified when it received the **B_GET_GRAPHICS_CARD_INFO** request.

---

draw_rect_with_8_bit_depth()

Index: 5

```
int32 draw_rect_with_8_bit_depth(int32 left, int32 top, int32 right, int32 bottom,
    uint8 colorIndex)
```

Tells the driver to fill a rectangle, specified by the first four arguments, with the color at *colorIndex* in the 8-bit color table. The arguments are frame buffer coordinates. The sides of the rectangle should be included in the area being filled.

---

draw_rect_with_32_bit_depth()

Index: 6

```
int32 draw_rect_with_32_bit_depth(int32 left, int32 top, int32 right, int32 bottom,
    uint32 color)
```

This is the same as **draw_rect_with_8_bit_depth()** except for the *color* argument. Here, *color* is a 32-bit value with 8-bit red, green, blue, and alpha components. The components are arranged in the order that the driver specified when it received the **B_GET_GRAPHICS_CARD_INFO** request.

See also: "B_GET_GRAPHICS_CARD_INFO, graphics_card_info" on page7

---

blit()

Index: 7

```
int32 blit(int32 sourceX, int32 sourceY,
    int32 destinationX, int32 destinationY,
    int32 width, int32 height)
```

Tells the driver to copy pixel data from a source rectangle to a destination rectangle. All coordinates and sizes are in frame buffer space. The left top pixel of the source rectangle is at (*sourceX*, *sourceY*) in the frame buffer. The left top pixel of the destination rectangle is at (*destinationX*, *destinationY*) in the frame buffer. Both rectangles are *width* pixels wide and *height* pixels high. The *width* and *height* arguments will always contain positive values, and the rectangles are guaranteed to lie wholly within the frame buffer.

---

draw_array_with_8_bit_depth() , indexed_color_line

Index: 8

```
int32 draw_array_with_8_bit_depth(indexed_color_line *array, int32 numItems,
```

bool *clipToRect*,
int16 *clipLeft*, int16 *clipTop*, int16 *clipRight*, int16 *clipBottom*)

Tells the driver to draw an array of lines in 8-bit depth. The line *array* holds a total of *numItems*. Each item is specified as an **indexed_color_line** structure, which contains the following fields (all coordinates are in frame buffer space):

int16 **x1**	The *x* coordinate of one end of the line.
int16 **y1**	The *y* coordinate of one end of the line.
int16 **x2**	The *x* coordinate of the other end of the line.
int16 **y2**	The *y* coordinate of the other end of the line.
uint8 **color**	The color of the line, expressed as an index into the color map.

If *clipToRect* is **true**, the function should draw only the portions of the lines that lie within the clipping rectangle defined by the last four arguments. The sides of the rectangle are included in the drawing area. If *clipToRect* is **false**, the final four arguments should be ignored

The lines should be minimally thin, as described under "draw_line_with_8_bit_depth()" on page20.

draw_array_with_32_bit_depth() , rgb_color_line

Index: 9

int32 **draw_array_with_32_bit_depth**(rgb_color_line **array*, int32 *numItems*,
bool *clipToRect*,
int16 *clipLeft*, int16 *clipTop*, int16 *clipRight*, int16 *clipBottom*)

Except for the color specification, which is encoded in the `rgb_color_line` structure, this is the same as **draw_array_with_8_bit_depth()**. The `rgb_color_line` structure contains these fields:

int16 **x1**	The *x* coordinate of one end of the line.
int16 **y1**	The *y* coordinate of one end of the line.
int16 **x2**	The *x* coordinate of the other end of the line.
int16 **y2**	The *y* coordinate of the other end of the line.
rgb_color **color**	The color of the line, expressed as a full 32-bit value.

sync()

Index: 10

int32 **sync**(void)

The driver should implement this function to block until all other currently-executing hook functions have finished. (More accurately, you only have to wait for those hook functions that actually touch the frame buffer.) The return value is ignored.

You should only implement this function if the card can perform any of the hook functions asynchronously. If all hook functions are synchronous, you should set the index 10 function to **NULL**.

After receiving a `sync()` call, your driver won't receive anymore hook functions until `sync()` returns. Thus, you don't have to guard against in-coming hook functions while sitting in `sync()`.

---

invert_rect()

Index: 11

```
int32 invert_rect(int32 left, int32 top, int32 right, int32 bottom)
```

Tells the driver to invert the colors in the rectangle specified by the arguments. The sides of the rectangle are included in the inversion.

---

draw_line_with_16_bit_depth()

Index: 12

```
int32 draw_line_with_16_bit_depth(int32 startX, int32 endX,
    int32 startY, int32 endY,
    uint16 color,
    bool clipToRect,
    int16 clipLeft, int16 clipTop, int16 clipRight, int16 clipBottom)
```

This is the same as `draw_rect_with_8_bit_depth()` except for the *color* argument. Here, *color* is a 16-bit value with red, green, blue, and (possibly) alpha components. The components are arranged in the order that the driver specified when it received the **B_GET_GRAPHICS_CARD_INFO** request.

---

draw_rect_with_16_bit_depth()

Index: 13

```
int32 draw_rect_with_16_bit_depth(int32 left, int32 top, int32 right, int32 bottom,
    uint16 color)
```

This is the same as `draw_rect_with_8_bit_depth()` except for the *color* argument. Here, *color* is a 16-bit value with red, green, blue, and (possibly) alpha components. The components are arranged in the order that the driver specified when it received the **B_GET_GRAPHICS_CARD_INFO** request.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BWindowScreen

Derived from: public BWindow
Declared in: be/game/WindowScreen.h
Library: libgame.so
Summary: *more...*

A BWindowScreen object provides exclusive access to the entire screen, bypassing the Application Server's window system. The object has direct access to the graphics card driver: It can set up the graphics environment on the graphics card, call driver-implemented drawing functions, and directly manipulate the frame buffer.

Screen Access

Like all windows, a BWindowScreen is hidden (off-screen) when it's constructed. By calling `Show()` to put it on-screen and make it the active window, an application takes over the whole screen. While the BWindowScreen is active, the Application Server's graphics operations are suspended—this means that you can't use any BView functions, nor any functions in classes derived from BView; you have to draw directly into the screen's frame buffer, and nothing except what the application draws will be visible to the user—no other windows and no desktop. When the BWindowScreen gives up active status, the Application Server automatically refreshes the screen with its old contents.

Although the BWindowScreen object provides a connection to the screen, you shouldn't draw from the BWindowScreen's thread. Use the thread only to regulate the access of other threads to the frame buffer.

Keyboard and Mouse

A BWindowScreen object remains a window while it has control of the screen; it stays attached to the Application Server and its message loop continues to function. It gets messages reporting the user's actions on the keyboard and mouse, just like any other active window. Because it covers the whole screen, it's notified of all mouse and keyboard events. You can attach filters to the window to get the messages as they arrive. Or you can call the Interface Kit's `get_key_info()` function to poll the state of the keyboard and construct a nominal BView so that you can call `GetMouse()` to poll the mouse.

Workspaces

This class respects workspaces. A BWindowScreen object releases its grip on the screen when the user turns to another workspace and reestablishes its control when the user returns to the workspace in which it's the active window.

Debugging

A BWindowScreen object can be constructed in a debugging mode that lets you switch back and forth between the workspace in which the game is running and a workspace where error messages are printed. See the constructor and the **RegisterThread()** function for details.

Hook Functions

ScreenConnected()

Can be implemented to do whatever is necessary when the BWindowScreen object obtains direct access to the frame buffer for the screen, and when it loses that access.

Constructor and Destructor

BWindowScreen()

```
BWindowScreen(const char *title, uint32 space, status_t *error,
    bool debugging = false)
```

Initializes the BWindowScreen object by assigning the window a *title* and specifying a *space* configuration for the screen. The window won't have a visible border or a tab in which to display the title to the user. However, others—such as the Workspaces application—can use the title to identify the window.

The window is constructed to fill the screen; its frame rectangle contains every screen pixel when the screen is configured according to the *space* argument. That argument describes the pixel dimensions and bits-per-pixel depth of the screen that the BWindowScreen object should establish when it obtains direct access to the frame buffer. It should be one of the following constants:

B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

These are the same constants that can be passed to **set_screen_space()**, the Interface Kit function that preference applications call to configure the screen.

The space configuration applies only while the BWindowScreen object is in control of the screen. When it gives up control, the previous configuration is restored.

The constructor assigns the window to the active workspace (**B_CURRENT_WORKSPACE**). It fails if another BWindowScreen object in any application is already assigned to the same workspace.

To be sure there wasn't an error in constructing the object, check the *error* argument. If all goes well, the constructor sets the *error* variable to **B_OK**. If not, it sets it to **B_ERROR**. If there's an error, it's likely to occur in this constructor, not the inherited BWindow constructor. Since the underlying window will probably exist, you'll need to instruct it to quit. For example:

```
status_t error;
MyWindowScreen *screen =
            new MyWindowScreen("Glacier", B_8_BIT_1024x768, &error);
if ( error != B_OK )
   screen->PostMessage(B_QUIT_REQUESTED, screen);
```

If the *debugging* flag is `true`, the BWindowScreen is constructed in debugging mode. This modifies its behavior and enables three functions, **`RegisterThread()`**, **`Suspend()`**, and **`SuspensionHook()`**. The debugging regime is described under those functions.

See also: **`RegisterThread()`**, the BScreen class in the Interface Kit

~BWindowScreen()

virtual **~BWindowScreen()**

Closes the clone of the graphics card driver (through which the BWindowScreen object established its connection to the screen), unloads it from the application, and cleans up after it.

Member Functions

CanControlFrameBuffer()

bool **CanControlFrameBuffer**(void)

Returns `true` if the graphics card driver permits applications to control the configuration of the frame buffer, and `false` if not. Control is exercised through these two functions:

- `SetFrameBuffer()`
  `MoveDisplayArea()`

A return of `true` means that these functions can communicate with the graphics card driver and at least the first will do something useful. A return of `false` means that neither of them will work.

See also: `SetFrameBuffer()`, `MoveDisplayArea()`

CardHookAt()

graphics_card_hook **CardHookAt**(int32 *index*)

Returns a pointer to the graphics card "hook" function that's located at *index* in its list of hook functions. The function returns `NULL` if the graphics card driver doesn't implement a function at that index or the index is out of range.

The hook functions provide accelerated drawing capabilities. They're documented under "Graphics Card Hook Functions". The first three hook functions (indices 0, 1, and 2) are not available through the Game Kit; if you pass an index of 0, 1, or 2 to `CardHookAt()`, it will return **`NULL`** even if the function is implemented.

An application can cache the pointers that `CardHookAt()` returns, but it should ask for a new set each time the depth or dimensions of the screen changes and each time the BWindowScreen object releases or regains control of the screen. You'd typically call `CardHookAt()` in your implementation of **`ScreenConnection()`**.

CardInfo()

graphics_card_info ***CardInfo**(void)

Returns a description of the current configuration of the graphics card, as kept by the driver for the card. The returned `graphics_card_info` structure is defined in **be/addons/graphics/GraphicsCard.h** and is documented in "The Entry Point and General Opcodes" in the "Graphics Card Drivers" chapter.

The information returned by this function is only valid when the BWindowScreen is connected to the display.

See also: `FrameBufferInfo()`

ColorList() see SetColorList()

Disconnect()

```
void Disconnect(void)
```

Forces the BWindowScreen object to disconnect itself from the screen—to give up its authority over the graphics card driver, allowing the Application Server to reassert control. Normally, you'd disconnect the BWindowScreen only when hiding the game, reducing it to an ordinary window in the background, or quitting. The `Hide()` and `Quit()` functions automatically disconnect the BWindowScreen as part of the process of hiding and quitting. `Disconnect()` allows you to sever the connection before calling those functions.

Before breaking the screen connection, `Disconnect()` causes the BWindowScreen object to receive a `ScreenConnected()` notification with a flag of `false`. It doesn't return until `ScreenConnected()` returns and the connection is broken. `Hide()` and `Quit()` share this behavior.

See also: `Hide()`, `Quit()`

FrameBufferInfo()

```
frame_buffer_info *FrameBufferInfo(void)
```

Returns a pointer to the `frame_buffer_info` structure that holds the driver's current conception of the frame buffer. The structure is defined in **addons/graphics/GraphicsCard.h** and is documented in "Frame Buffer Opcodes" in the "Graphics Card Drivers" chapter.

The information returned by this function is only valid if `SetFrameBuffer()` has been called.

See also: `SetSpace()`, `SetFrameBuffer()`, `MoveDisplayArea()`, `CardInfo()`

Hide() , Show()

```
virtual void Hide(void)
virtual void Show(void)
```

These functions augment their BWindow counterparts to make sure that the BWindowScreen is disconnected from the screen before it's hidden and that it's ready to establish a connection

when it becomes the active window.

Hide() calls ScreenConnected() (with an argument of false) and breaks the connection to the screen when ScreenConnected() returns. It then hides the window.

Show() shows the window on-screen and makes it the active window, which will cause it to establish a direct connection to the graphics card driver for the screen. Unlike Hide(), it may return before ScreenConnected() is called (with an argument of true).

See also: BWindow::Hide()

IOBase()

```
void *IOBase(void)
```

Returns a pointer to the base address for the input/output registers on the graphics card. Registers are addressed by 16-bit offsets from this base address. (This function may not be supported in future releases.)

MoveDisplayArea()

```
status_t MoveDisplayArea(int32 x, int32 y)
```

Relocates the display area, the portion of the frame buffer that's mapped to the screen. This function moves the area's left-top corner to (*x*, *y*); by default, the corner lies at (0, 0). The display area must lie entirely within the frame buffer.

MoveDisplayArea() only works if the graphics card driver permits application control over the frame buffer. It must also permit a frame buffer with a total area larger than the display area. If successful in relocating the display area, this function returns B_OK; if not, it returns B_ERROR.

See also: CanControlFrameBuffer()

Quit()

```
virtual void Quit(void)
```

Augments the BWindow version of Quit() to force the BWindowScreen object to disconnect itself from the screen, so that it doesn't quit while in control of the frame buffer.

Although Quit() disconnects the object before quitting, this may not be soon enough for your application. For example, if you need to destroy some drawing threads before the BWindowScreen object is itself destroyed, you should get rid of them after the screen connection is severed. You can force the object to disconnect itself by calling Disconnect(). For example:

```
```

```
void MyWindowScreen::Quit()
{
   Disconnect();
   kill_thread(drawing_thread_a);
   kill_thread(drawing_thread_b);
   BWindowScreen::Quit();
}
```

If the screen connection is still in place when **Quit()** is called, it calls **ScreenConnected()** with a flag of `false`. It doesn't return until **ScreenConnected()** returns and the connection is broken.

See also: **ScreenConnected()**

RegisterThread() , Suspend() , SuspensionHook()

void **RegisterThread(**thread_id *thread***)**
void **Suspend(**char **label***)**
virtual void ***SuspensionHook(**bool *suspended***)**

These three functions aid in debugging a game application. They have relevance only if the BWindowScreen is running in debugging mode. To set up the mode, you must:

- Construct the BWindowScreen with the *debugging* flag set to `true`. The flag is `false` by default.
- Register all drawing threads (all threads that can touch the frame buffer in any way) by passing the **thread_id** to **RegisterThread()** immediately after the thread is spawned—before **resume_thread()** is called to start the thread's execution. The window thread for the BWindowScreen object should not draw and should not be registered.
- Launch the application from the command line in a Terminal window. The window will collect debugging output from the application while the BWindowScreen runs in a different workspace, generally the one at the immediately preceding index. For example, if the Terminal window is in the fifth workspace (Command-F5), the game will run in the fourth (Command-F4); if the Terminal is in the fourth (CommandF4), the game runs in the third (Command-F3); and so on. However, if the Terminal window is in the first workspace (Command-F1), the game runs in the second (Command-F2).

The Terminal window is the destination for all messages the game writes to the standard error stream or to the standard output—from **printf()**, for example. You can switch back and forth between the game and Terminal workspaces to check the messages and run your application. When you switch from the game workspace to the Terminal workspace, all registered threads are suspended and the graphics context is saved. When you switch back to the game, the graphics context is restored and the threads are resumed.

Calling **Suspend()** switches to the Terminal workspace programmatically, just as pressing the correct Command–function key combination would. Registered threads are suspended, the Terminal workspace is activated, and the *label* passed as an argument is displayed in a

message in the Terminal window. You can resume the game by manually switching back to its workspace.

`SuspensionHook()` is called whenever the game is suspended or resumed—whether by the user switching workspaces or by `Suspend()`. It gives you an opportunity to save and restore any state that would otherwise be lost. `SuspensionHook()` is called with a *suspended* flag of `true` just after the application is suspended and with a flag of `false` just before it's about to be resumed.

`ScreenConnected()` is *not* called when you switch between the Terminal and game workspaces while in debugging mode. However, it is called for all normal game activities—when the BWindowScreen is first activated and when it hides or quits, for example.

Debugging mode can also preserve some information in case of a crash. Hold down all the left modifier keys (Shift, Control, Option, Command, Alt, or whatever the keys may happen to be on your keyboard), and press the F12 key. This restarts the screen with a 640 * 480 resolution and displays a debugger window. You should then be able to switch to the Terminal workspace to check the last set of messages before the crash, modify your code, and start again.

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Overrides the BWindow version of `ScreenChanged()` so that it does nothing. This function is called automatically when the screen configuration changes. It's not one that you should call in application code or reimplement for the game.

See also: `BWindow::ScreenChanged()`

ScreenConnected()

virtual void **ScreenConnected(**bool *connected***)**

Implemented by derived classes to take action when the application gains direct access to the screen and when it's about to lose that access.

This function is called with the *connected* flag set to **`true`** immediately after the BWindowScreen object becomes the active window and establishes a direct connection to the graphics card driver for the screen. At that time, the Application Server's connection to the screen is suspended; drawing can only be accomplished through the screen access that the BWindowScreen object provides.

`ScreenConnected()` is called with a flag of `false` just before the BWindowScreen object is scheduled to lose its control over the screen and the Application Server's control is reasserted. The BWindowScreen's connection to the screen will not be broken until this function returns.

It should delay returning until the application has finished all current drawing and no longer needs direct screen access.

Note that whenever `ScreenConnected()` is called, the BWindowScreen object is guaranteed to be connected to the screen; if *connected* is `true`, it just became connected, if *connected* is `false`, it's still connected but will be disconnected when the function returns.

Derived classes typically use this function to regulate access to the screen. For example, they may acquire a semaphore when the *connected* flag is `false`, so that application threads won't attempt direct drawing when the connection isn't in place, and release the semaphore for drawing threads to acquire when the flag is `true`. For example:

```
void MyWindowScreen::ScreenConnected(bool connected)
{
   if ( connected == false )
      acquire_sem(directDrawingSemaphore);
   else
      release_sem(directDrawingSemaphore);
}
```

See also: `Disconnect()`

SetColorList() , ColorList()

void **SetColorList(**rgb_color **colors*, int32 *first* = 0, int32 *last* = 255**)**

rgb_color ***ColorList(**void**)**

These functions set and return the list of 256 colors that can be displayed when the frame buffer has a depth of 8 bits per pixel (the `B_CMAP8` color space). `SetColorList()` is passed an array of one or more *colors* to replace the colors currently in the list. The first color in the array replaces the color in the list at the specified *first* index; all colors up through the *last* specified index are modified. It fails if either index is out of range.

`SetColorList()` alters the list of colors kept on the graphics card. If the BWindowScreen isn't connected to the screen, the new list takes effect when it becomes connected.

`ColorList()` returns a pointer to the entire list of 256 colors. This is not the list kept by the graphics card driver, but a local copy. It belongs to the BWindowScreen object and should be altered only by calling `SetColorList()`.

See also: `BScreen::ColorMap()` in the Interface Kit

SetFrameBuffer()

status_t **SetFrameBuffer(**int32 *width*, int32 *height***)**

Configures the frame buffer on the graphics card so that it's *width* pixel columns wide and *height* pixel rows high. This function works only if the driver for the graphics card allows custom configurations (as reported by **CanControlFrameBuffer()**) and the BWindowScreen object is currently connected to the screen.

The new dimensions of the frame buffer must be large enough to hold all the pixels displayed on-screen—that is, they must be at least as large as the dimensions of the display area. If the driver can't accommodate the proposed *width* and *height*, **SetFrameBuffer()** returns **B_ERROR**. If the change is made, it returns **B_OK**.

This function doesn't alter the depth of the frame buffer or the size or location of the display area.

See also: **MoveDisplayArea()**, **SetSpace()**

SetSpace()

status_t **SetSpace(**uint32 *space***)**

Configures the screen space to one of the standard combinations of width, height, and depth. The configuration is first set by the class constructor—permitted *space* constants are documented there—and it may be altered after construction only by this function.

Setting the screen space sets the dimensions of the frame buffer and display area. For example, if *space* is **B_32_BIT_800x600**, the frame buffer will be 32 bits deep and at least 800 pixel columns wide and 600 pixel rows high. The display area (the area of the frame buffer mapped to the screen) will also be 800 pixels * 600 pixels. After setting the screen space, you can enlarge the frame buffer by calling **SetFrameBuffer()** and relocate the display area in the larger buffer by calling **MoveDisplayArea()**.

If the requested configuration is refused by the graphics card driver, **SetSpace()** returns **B_ERROR**. If all goes well, it returns **B_OK**.

See also: the BWindowScreen constructor, **SetFrameBuffer()**, **MoveDisplayArea()**

Suspend() see RegisterThread()

SuspensionHook() see RegisterThread()

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Overrides the BWindow version of **WindowActivated()** to connect the BWindowScreen object to the screen (give it control over the graphics card driver) when the *active* flag is `true`.

This function doesn't disconnect the BWindowScreen when the flag is `false`, because there's no way for the window to cease being the active window without the connection already having been lost.

Don't reimplement this function in your application, even if you call the inherited version; rely instead on `ScreenConnected()` for accurate notifications of when the BWindowScreen gains and loses control of the screen.

See also: `BWindow::WindowActivated()`, `ScreenConnected()`

WorkspaceActivated()

virtual void **WorkspaceActivated(**int32 *workspace*, bool *active***)**

Overrides the BWindow version of `WorkspaceActivated()` to connect the BWindowScreen object to the screen when the *active* flag is `true` and to disconnect it when the flag is `false`. User's typically activate the game by activating the workspace in which it's running, and deactivate it by moving to another workspace.

Don't override this function in your application; implement `ScreenConnected()` instead.

See also: `BWindow::WorkspaceActivated()`, `ScreenConnected()`

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BWindowScreen Function Summary

Function Summary

Function	Description
CanControlFrameBuffer()	Tests whether the graphics card driver allows application control of the frame buffer
CardHookAt()	Returns pointer to a given graphics hook function
CardInfo()	Returns a description of the current graphics card configuration
ColorList()	Returns the 8-bit color map
Disconnect()	Forces the object to disconnect from the screen
FrameBufferInfo()	Returns a pointer to a description of the frame buffer
Hide()	Provided for implementation
IOBase()	Returns a pointer to the card's I/O registers
MoveDisplayArea()	Relocates the display area
Quit()	Provided for implementation
RegisterThread()	Registers a drawing thread for debugging
ScreenChanged()	Provided for implementation
ScreenConnected()	Hook called when application gains or loses access to the screen

SetColorList()	Sets the 8-bit color map
SetFrameBuffer()	Sets the frame buffer to a nonstandard dimension and resolution
SetSpace()	Sets the frame buffer to a standard dimension and resolution
Show()	Provided for implementation
Suspend()	Switches the workspace for debugging
SuspensionHook()	Hook called when the workspace is changed to or from the application's workspace
WindowActivated()	Provided for implementation
BWindowScreen()	Constructor
~BWindowScreen()	Destructor
WorkspaceActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Input-Related Functions

The following functions are the BView non-hook function related to user input and input processing. Note that the most important input-related functions are hook functions; see the Hook Functions section.

BeginRectTracking() , EndRectTracking()

void **BeginRectTracking(**BRect *rect*, uint32 *how* = B_TRACK_WHOLE_RECT**)**
void **EndRectTracking(**void**)**

These functions instruct the Application Server to display a rectangular outline that will track the movement of the cursor. **`BeginRectTracking()`** puts the rectangle on-screen and initiates tracking; **`EndRectTracking()`** terminates tracking and removes the rectangle. The initial rectangle, *rect*, is specified in the BView's coordinate system.

This function supports two kinds of tracking, depending on the constant passed as the *how* argument:

B_TRACK_WHOLE_RECT	The whole rectangle moves with the cursor. Its position changes, but its size remains fixed.
B_TRACK_RECT_CORNER	The left top corner of the rectangle remains fixed within the view while its right and bottom edges move with the cursor.

Tracking is typically initiated from within a BView's **`MouseDown()`** function and is terminated in **`MouseUp()`**

DragMessage()

void **DragMessage(**BMessage **message*, BBitmap **image*, BPoint *point*, BHandler **replyTarget* = NULL**)**
void **DragMessage(**BMessage **message*, BRect *rect*, BHandler **replyTarget* = NULL**)**
void **DragMessage(**BMessage **message*, BBitmap **image*, drawing_mode *dragMode*, BPoint *offset*, BHandler **replyTarget* = NULL**)**

Initiates a drag-and-drop session.

message, is a BMessage object that bundles the information that will be dragged and dropped on the destination view. The caller retains responsibility for this object and can delete it after **`DragMessage()`** return (the BView makes a copy of the message).

image, is a bitmap that the user can drag. The bitmap is automatically freed when the message is dropped.

1 bit-per-pixel bitmaps aren't supported; you should avoid using them.

point locates the hotspot within *image* (in the bitmap's coordinate system). This is the point that's aligned with the location passed to **MouseDown()** or returned by **GetMouse()**.

rect defines the dimensions of an outline rectangle that you can instead of a bitmap. The rectangle is stated in the BView's coordinate system.

replyTarget, names the object that you want to handle a message that might be sent in reply to the dragged message. If *replyTarget* is **NULL**, as it is by default, any reply that's received will be directed to the BView object that initiated the drag-and-drop session.

dragMode defines the **drawing_mode** which will be used to draw *image* as the image is dragged around. This is provided primarily so that transparent or partially transparent images can be dragged around (using the **B_OP_ALPHA** drawing mode).

This function works only for BView objects that are attached to a window.

EndRectTracking() see BeginRectTracking()

GetMouse()

void **GetMouse(**BPoint ***cursor**, uint32 ***buttons**, bool *checkQueue* = true**)**

Provides the location of the cursor and the state of the mouse buttons. The position of the cursor is recorded in the variable referred to by *cursor*; it's provided in the BView's own coordinates. A bit is set in the variable referred to by *buttons* for each mouse button that's down. This mask may be 0 (if no buttons are down) or it may contain one or more of the following constants:

B_PRIMARY_MOUSE_BUTTO N
B_SECONDARY_MOUSE_BU TTON
B_TERTIARY_MOUSE_BUTT ON

The cursor doesn't have to be located within the view for this function to work; it can be anywhere on-screen. However, the BView must be attached to a window.

If the *checkQueue* flag is set to `false`, **GetMouse()** provides information about the current state of the mouse buttons and the current location of the cursor.

If *checkQueue* is `true`, as it is by default, this function first looks in the message queue for any pending reports of mouse-moved or mouse-up events. If it finds any, it takes the one that has been in the queue the longest (the oldest message), removes it from the queue, and reports the *cursor* location and *button* states that were recorded in the message. Each **GetMouse()** call removes another message from the queue. If the queue doesn't hold any **B_MOUSE_MOVED** or

B_MOUSE_UP messages, GetMouse() reports the current state of the mouse and cursor, just as if *checkQueue* were `false`.

If *checkQueue* is **true**, and the view's parent window has pending update events, GetMouse() causes those update events to be processed.

You shouldn't use this function to track the mouse; implement the MouseMoved() function instead.

See also: modifiers()

MakeFocus()

virtual void **MakeFocus(**bool *focused* = true**)**

Makes the BView the current focus view for its window (if the *focused* flag is `true`), or causes it to give up that status (if *focused* is `false`). The focus view is the view that displays the current selection and is expected to handle reports of key-down events when the window is the active window. There can be no more than one focus view per window at a time.

When called to make a BView the focus view, this function invokes MakeFocus() for the previous focus view, passing it an argument of `false`. It's thus called twice—once for the new and once for the old focus view.

Calling MakeFocus() is the only way to make a view the focus view; the focus doesn't automatically change on mouse-down events. BViews that can display the current selection (including an insertion point) or that can accept pasted data should call MakeFocus() in their MouseDown() functions.

A derived class can override MakeFocus() to add code that takes note of the change in status. For example, a BView that displays selectable data may want to highlight the current selection when it becomes the focus view, and remove the highlighting when it's no longer the focus view. A BView that participates in the keyboard navigation system should visually indicate that it can be operated from the keyboard when it becomes the focus view, and remove that indication when the user navigates to another view and it's notified that it's no longer the focus view.

If the BView isn't attached to a window, this function has no effect.

See also: BWindow::CurrentFocus(), IsFocus()

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Augments the BHandler version of MessageReceived() to handle scripting messages for the BView.

See also: `BHandler::MessageReceived()`

ScrollBar()

BScrollBar ***ScrollBar(**orientation *posture***)** const

Returns a BScrollBar object that scrolls the BView (that has the BView as its target). The requested scroll bar has the *posture* orientation—`B_VERTICAL` or `B_HORIZONTAL`. If the BView isn't the target of a scroll bar with the specified orientation, this function returns `NULL`.

See also: `ScrollBar::SetTarget()`

ScrollBy() , ScrollTo()

void **ScrollBy(**float *horizontal*, float *vertical***)**

virtual void **ScrollTo(**BPoint *point***)**
inline void **ScrollTo(**float *x*, float *y***)**

These functions scroll the contents of the view, provided that the BView is attached to a window.

`ScrollBy()` adds *horizontal* to the left and right components of the BView's bounds rectangle, and *vertical* to the top and bottom components. This serves to shift the display *horizontal* coordinate units to the left and *vertical* units upward. If *horizontal* and *vertical* are negative, the display shifts in the opposite direction.

`ScrollTo()` shifts the contents of the view as much as necessary to put *point*—or (*x*, *y*)—at the upper left corner of its bounds rectangle. The point is specified in the BView's coordinate system.

Anything in the view that was visible before scrolling and also visible afterwards is automatically redisplayed at its new location. The remainder of the view is invalidated, so the BView's `Draw()` function will be called to fill in those parts of the display that were previously invisible. The update rectangle passed to `Draw()` will be the smallest possible rectangle that encloses just these new areas. If the view is scrolled in only one direction, the update rectangle will be exactly the area that needs to be drawn.

If the BView is the target of scroll bars, `ScrollBy()` and `ScrollTo()` notify the BScrollBar objects of the change in the display so they can update themselves to match. If the contents were scrolled horizontally, they call the horizontal BScrollBar's `SetValue()` function and pass it the new value of the left side of the bounds rectangle. If they were scrolled vertically, they call `SetValue()` for the vertical BScrollBar and pass it the new value of the top of the bounds rectangle.

The inline version of `ScrollTo()` works by creating a BPoint object and passing it to the version that's declared `virtual`. Therefore, if you want to override either function, you should override the virtual version. (However, due to the peculiarities of C++, overriding any version of an

overloaded function hides all versions of the function. For continued access to the nonvirtual version without explicitly specifying the "BView::" prefix, simply copy the inline code from **interface/View.h** into the derived class.)

SetEventMask() , SetMouseEventMask() , EventMask()

status_t **SetEventMask(**uint32 *events*, uint32 *options*=0**)**
status_t **SetMouseEventMask(**uint32 *events*, uint32 *options*=0**)**
uint32 **EventMask(**void**)**

`SetEventMask()` lets you extend the scope of the mouse and keyboard events that the view can receive. If *events* include **`B_POINTER_EVENTS`**, the view will receive *mouse events* (aka *pointer events*) even when the mouse isn't over the view; if it includes **`B_KEYBOARD_EVENTS`**, the view will receive *keyboard events* even if the view isn't in *focus*. (We'll look at the *options* argument below).

`SetMouseEventMask()` does the same thing as `SetEventMask()`, except **(1)** it can only be called from within an implementation of `MouseDown()`, and **(2)** the new *events* value—which is added to the current event mask—is only in effect until (and including) the following mouse up event. When the mouse is released, the view's previous event mask (as set through `SetEventMask()`) is re-established.

The *option* arguments lets you request other event-handling modifications (note that `SetEventMask()` only accepts the first of these options; `SetMouseEventMask()` accepts all three):

- **`B_NO_POINTER_HISTORY`**. This tells the App Server to only send the most recent pointer moved (i.e. mouse moved) event to your view (all "old" events are thrown away). You use this option if your `MouseMoved()` implementation is too heavy to keep up with the mouse moved messages that are pouring in. Of course, your view may lose some mouse movement granularity, but that's the price you pay to stay in synch with the user.
- **`B_SUSPEND_VIEW_FOCUS`** (*SetMouseEventMask() only*). Events that are normally sent to the focus view are suppressed. In practice, this means that while the mouse is held down, the keyboard is turned off. Note that the view that's processing the `MouseDown()` messages doesn't have to be the focus view to suppress focused messages.
- **`B_LOCK_WINDOW_FOCUS`** (*SetMouseEventMask() only*). Prevents the view's window from losing focused status while the mouse is down, even if the mouse leaves the window's bounds.

To ask for an option without changing the event mask (or mouse event mask), pass 0 as the *events* argument.

Both `SetEventMask()` and `SetMouseEventMask()` require that the view be attached to a window; they have no effect if the view isn't already attached.

`EventMask()` returns the view's event mask as set through `SetEventMask()`. It doesn't consider the mask set in `SetMouseEventMask()`.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BButton

Derived from: public BControl
Declared in: be/interface/Button.h
Library: libbe.so
Summary: *more...*

A BButton object draws a labeled button on-screen and responds when the button is clicked or when it's operated from the keyboard. If the BButton is the *default button* for its window and the window is the active window, the user can operate it by pressing the Enter key.

BButtons have a single state. Unlike check boxes and radio buttons, the user can't toggle a button on and off. However, the button's value changes while it's being operated. During a click (while the user holds the mouse button down and the cursor points to the button on-screen), the BButton's value is set to 1 (`B_CONTROL_ON`). Otherwise, the value is 0 (`B_CONTROL_OFF`).

This class depends on the control framework defined in the BControl class. In particular, it calls these BControl functions:

- `SetValue()` to make each change in the BControl's value. This is a hook function that you can override to take collateral action when the value changes.
- `Invoke()` to post a message each time the button is clicked or operated from the keyboard. You can designate the object that should handle the message by calling BControl's `SetTarget()` function. A model for the message is set by the BButton constructor (or by BControl's `SetMessage()` function).
- `IsEnabled()` to determine how the button should be drawn and whether it's enabled to post a message. You can call BControl's `SetEnabled()` to enable and disable the button.

A BButton is an appropriate control device for initiating an action. Use a BCheckBox, a BPictureButton, or BRadioButtons to set a state.

Hook Functions

MakeDefault()
Makes the BButton the default button for its window or removes that status; can be augmented by derived classes to take note when the status of the button changes.

Constructor and Destructor

BButton()

```
BButton(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BButton(BMessage *archive)
```

Initializes the BButton by passing all arguments to the BControl constructor. BControl initializes the button's *label* and assigns it a model *message* that identifies the action that should be carried out when the button is invoked.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change.

When the button is attached to a window, it will be resized to its preferred height; the height of BButton's *frame* rectangle will exactly accommodate the button border and label, given the BButton's current font.

See also: the BControl and BView constructors, **`BControl::Invoke()`**

~BButton()

```
virtual ~BButton()
```

Does nothing; a BButton has no data to free.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow**(void)

Augments the BControl version of this function to tell the BWindow that the button is the default button, if `MakeDefault()` has already been called.

See also: `BView::AttachedToWindow()`, `BControl::AttachedToWindow()`, `MakeDefault()`

Draw()

virtual void **Draw**(BRect *updateRect*)

Draws the button and labels it. If the BButton's value is anything but 0, the button is highlighted. If it's disabled, it drawn in muted shades of gray. Otherwise, it's drawn in its ordinary, enabled, unhighlighted state.

See also: `BView::Draw()`

GetPreferredSize()

virtual void **GetPreferredSize**(float **width*, float **height*)

Calculates how big the button needs to be to display its label in the current font, and writes the results into the variables that the *width* and *height* arguments refer to. `ResizeToPreferred()`, defined in the BView class, resizes a view's frame rectangle to the preferred size, keeping its left and top sides constant. A button is automatically resized to its preferred height (but not to its preferred width) by `AttachedToWindow()`.

MakeDefault

KeyDown()

virtual void **KeyDown**(const char **bytes*, int32 *numBytes*)

Augments the inherited version of `KeyDown()`MakeDefault to respond to messages reporting that the user pressed the Enter key or the space bar. Its response is to:

- Momentarily highlight the button and change its value.

- Call **Invoke()** to deliver a copy of the model BMessage to the target receiver.

The BButton gets **KeyDown()** function calls when it's the focus view for the active window (which results when the user navigates to it) and also when it's the default button for the window and the character the user types is **B_ENTER**.

See also: **BControl::Invoke()**, **BView::KeyDown()**, **MakeDefault()**

MakeDefault() , IsDefault()

virtual void **MakeDefault(**bool *flag***)**
bool **IsDefault(**void**)** const

MakeDefault() makes the BButton the default button for its window when *flag* is `true`, and removes that status when *flag* is `false`. The default button is the button the user can operate by striking the Enter key when the window is the active window. **IsDefault()** returns whether the BButton is currently the default button.

A window can have only one default button at a time. Setting a new default button, therefore, may deprive another button of that status. When **MakeDefault()** is called with an argument of `true`, it generates a **MakeDefault()** call with an argument of `false` for previous default button. Both buttons are redisplayed so that the user can see which one is currently the default.

The default button can also be set by calling BWindow's **SetDefaultButton()** function. That function makes sure that the button that's forced to give up default status and the button that obtains it are both notified through **MakeDefault()** function calls.

MakeDefault() is therefore a hook function that can be augmented to take note each time the default status of the button changes. It's called once for each change in status, no matter which function initiated the change.

See also: **BWindow::SetDefaultButton()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **MouseDown()** to track the cursor while the user holds the mouse button down. As the cursor moves in and out of the button, the BButton's value is reset accordingly. The **SetValue()** virtual function is called to make the change each time.

If the cursor is inside the BButton's bounds rectangle when the user releases the mouse button, this function posts a copy of the model message so that it will be dispatched to the target object.

See also: **BView::MouseDown()**, **BControl::Invoke()**, **BControl::SetTarget()**

SetLabel()

virtual void **SetLabel(**const char ***string**)

Overrides the BControl version of this function to make sure that calculations based on the width of the label won't assume cached results for the previous label.

See also: `BControl::SetLabel()`

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_default"	`B_BOOL_TYPE`	Exists and is `true` if the button is the default.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BButton Function Summary

Function Summary

Function	Description
AttachedToWindow()	Provided for implementation
BButton()	Constructor
~BButton()	Destructor
Draw()	Provided for implementation
GetPreferredSize()	Provided for implementation
IsDefault()	Tests whether button is default for a window
KeyDown()	Provided for implementation
MakeDefault()	Sets or removes status as default button for a window
MouseDown()	Provided for implementation
SetLabel()	Sets button label

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

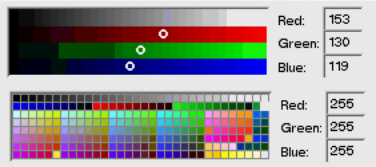
Copyright © 2000 Be, Inc. All rights reserved..

BColorControl

Derived from: BControl > BView > BHandler
Mix-in classes: BArchivable, BInvoker
Declared in: be/interface/ColorControl.h
Library: libbe.so
Summary: *more...*

A BColorControl object displays an on-screen device that permits users to pick a color. It reports the color as its current value—an **rgb_color** data structure stored as a 32-bit integer. If a model message is provided, it announces each change in value by sending a copy of the message to a designated target.

When the screen is 8 bits deep, the BColorControl object presents users with a matrix of the 256 available colors. The user chooses a color by pressing the primary mouse button while the cursor is over one of the cells in the matrix. Dragging from cell to cell changes the selected color. The arrow keys can similarly change the selection when the object is the focus view. The BColorControl's value changes each time the selection does.



When the screen is 15, 16, or 32 bits deep, the BColorControl object displays ramps for each color component. The user changes the current color by modifying a red, green, or blue component value.

In addition to the color matrix and ramp, a BColorColor has three text fields where the user can set a color by typing in its red, green, and blue component values. The text fields (BTextControl objects) are children of the BColorControl.

Constructor and Destructor

BColorControl() , color_control_layout

```
BColorControl(BPoint leftTop, color_control_layout matrix, float cellSide,
    const char *name, BMessage *message = NULL,
    bool bufferedDrawing = false)
BColorControl(BMessage *archive)
enum color_control_layout {}
```

Initializes the BColorControl so that the left top corner of its frame rectangle will be located at the stated *leftTop* point in the coordinate system of its parent view. The frame rectangle will be large enough to display 256 color cells arranged in the specified *matrix*, which can be any of the following `color_control_layout` constants:

B_CELLS_4x64	**B_CELLS_8x32**	**B_CELLS_16x16**
B_CELLS_64x4	**B_CELLS_32x8**	

For example, **B_CELLS_4x64** lays out a matrix with four cell columns and 64 rows; **B_CELLS_32x8** specifies 32 columns and 8 rows. Each cell is a square *cellSide* coordinate units on a side; since the number of units translates directly to screen pixels, *cellSide* should be a whole number.

When the screen is 15, 16, or 32 bits deep, the same frame rectangle will display four color ramps, one each for the red, green, and blue components, plus a disabled ramp for the alpha component. You might choose *matrix* and *cellSize* values with a view toward how the resulting bounds rectangle would be divided into four horizontal rows.

The *name* argument assigns a name to the object as a BHandler. It's the same as the argument declared by the BView constructor.

If a model *message* is supplied, the BColorControl will announce every change in color value by calling **Invoke()** (defined in the BControl class) to post a copy of the message to a designated target.

If the *bufferedDrawing* flag is `true`, all changes to the on-screen display will first be made in an off-screen bitmap and then copied to the screen. This makes the drawing smoother, but it requires more memory.

The initial value of the new object is 0, which when translated to an **rgb_color** structure, means black.

See also: **BHandler::SetName()**, **BControl::Invoke()**

~BColorControl()

virtual ~**BColorControl()**

Gets rid of the off-screen bitmap, if one was requested when the object was constructed.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BColorControl object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* doesn't contain data for a BColorControl object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, then adds the layout, cell size, and whether the object uses buffered drawing to the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of this function to set the BColorControl's view color and low color to be the same as its parent's view color and to set up the BTextControl objects where the user can type red, green, and blue color values. If the object uses buffered drawing, this function makes sure the offscreen images are displayed on-screen.

See also: **BControl::AttachedToWindow()**, **BView::SetViewColor()**

CellSize() see SetCellSize()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Overrides the BView version of this function to draw the color control.

See also: **BView::Draw()**

GetPreferredSize()

virtual void **GetPreferredSize(**float **width*, float **height***)**

Calculates how large the color control needs to be given its layout, cell size, and current font; the results are reported in the variables that the *width* and *height* arguments refer to.

See also: **BView::GetPreferredSize()**

KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes***)**

Augments the BControl version of **KeyDown()** to allow the user to navigate within the color control using the arrow keys.

See also: **BControl::KeyDown()**

Layout() see SetLayout()

MessageReceived()

virtual void **MessageReceived(**BMessage *message***)**

Responds to internal messages that change the color.

See also: **BHandler::MessageReceived()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **MouseDown()** to allow the user to operate the color control with the mouse.

See also: **BView::MouseDown()**

SetCellSize() , CellSize()

virtual void **SetCellSize(**float *cellSide***)**

float **CellSize(**void**)** const

These functions set and return the size of a single cell in the BColorControl's matrix of 256 colors. A cell is a square *cellSide* coordinate units on a side. The size is first set by the BColorControl constructor.

See also: the BColorControl constructor

SetEnabled()

virtual void **SetEnabled(**bool *enabled***)**

Auguments the BControl version of `SetEnabled()` to disable and re-enable the text fields for setting the color components as the BColorControl is disabled and re-enabled. The inherited `IsEnabled()` function doesn't need augmenting and therefore isn't reimplemented.

See also: `BControl::SetEnabled()`

SetLayout() , Layout()

virtual void **SetLayout(**color_control_layout *layout***)**
color_control_layout **Layout(**void**)** const

These functions set and return the layout of the matrix of 256 color cells. The matrix is first arranged by the constructor. See the constructor for permissible *layout* values.

See also: the BColorControl constructor

SetValue() , ValueAsColor()

virtual void **SetValue(**int32 *color***)** inline void **SetValue(**rgb_color *color***)**
rgb_color **ValueAsColor(**void**)**

These functions set and return the BColorControl's current value—the last color that the user selected.

The virtual version of `SetValue()` takes an `int32` argument and is essentially the same as the BControl version of the function, which it modifies only to take care of class-internal housekeeping details. The inline version, on the other hand, takes an `rgb_color` argument and is unique to this class. It packs color information from the structure into a 32-bit integer and passes it to the virtual version of the function. Like all other objects that derive from BControl, a BColorControl stores its current value as an `int32`; no information is lost in the translation from an `rgb_color` structure to an integer.

`SetValue()` is called to make every change to the BControl's value. If you override this function to be notified of the changes, you should override the virtual version. (However, due to the peculiarities of C++, overriding any version of an overloaded function hides all versions of the function. For continued access to the `rgb_color` version of `SetValue()` without explicitly specifying the "BColorControl::" prefix, copy the inline code from **interface/ColorControl.h** to the derived class.)

`ValueAsColor()` is an alternative to the `Value()` function inherited from the BControl class. It returns the object's current value as an `rgb_color`; `Value()` returns it as an `int32`.

See also: `BControl::SetValue()`

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_layout"	`B_INT32_TYPE`	The layout of the matrix of 256 color cells.
"_csize"	`B_FLOAT_TYPE`	Size of a single cell in the matrix of 256 color cells.
"_use_off"	`B_BOOL_TYPE`	Whether the control uses buffered drawing.

The following views are added to the "_views" field (deep copy only):

Name	Level	Meaning
"_red"	0	BTextControl for the "red" field.
"_green"	0	BTextControl for the "green" field.
"_blue"	0	BTextControl for the "blue" field.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BColorControl Function Summary

Function Summary

Function	Description
Archive()	Archives a BColorControl into a BMessage
AttachedToWindow()	Provided for implementation
CellSize()	Returns size of a single cell in the displayed palette
BColorControl()	Constructor
~BColorControl()	Destructor
Draw()	Provided for implementation
GetPreferredSize()	Provided for implementation
Instantiate()	Creates a BColorControl from a BMessage
KeyDown()	Provided for implementation
Layout()	Returns current layout of color palette
MessageReceived()	Provided for implementation
MouseDown()	Provided for implementation
SetCellSize()	Sets size of a single cell in the displayed palette
SetEnabled()	Provided for implementation
SetLayout()	Sets layout of color palette
SetValue()	Sets layout of color palette
ValueAsColor()	Returns currently selected color

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTextControl

Derived from: public BControl
Declared in: be/interface/TextControl.h
Library: libbe.so
Summary: *more...*

A BTextControl object displays a labeled text field that behaves like other control devices. When the user presses certain keys after modifying the text in the field, it delivers a message to a designated target.

There are two parts to the view: A static label on the left, which the user cannot modify, and an editable field on the right, which behaves just like a one-line BTextView. In fact, the BTextControl installs a BTextView object as its child to handle editing chores within this part of the view. It's this child view that responds to keyboard events for the BTextControl rather than the control object itself.

The child BTextView must become the focus view for the window before the user can enter or edit text in the field. If the user modifies the contents of the field and then causes the child to cease being the focus view, the BTextControl delivers a message to its target, just like any other BControl object when it's invoked. The message notifies the target that the user has finished making changes to the text. (It doesn't matter what causes the change in focus—a click in another text field, for example, or a `B_TAB` character that navigates to another view.)

The BTextControl is also invoked when the user types a `B_ENTER` character, though this doesn't change the focus view. It selects all the text in the field.

You can arrange for another message—a "modification message"—to be sent when the user makes the first change to the text after the child BTextView has become the focus view (or after `B_ENTER` caused all the text to be selected). This message notifies the target that editing has begun.

Note that BTextControls only allow the user to edit a single line of text; newline characters are automatically stripped from the text, if they're inserted.

Because the label is drawn by the BTextControl itself and the editable text is drawn by its child BTextView, you can assign different properties (color or font, for example) to each string. The BTextControl has only one child; `ChildAt()` returns it when passed an index of 0.

Constructor and Destructor

BTextControl()

```
BTextControl(BRect frame, const char *name,
    const char *label, const char *text,
    BMessage *message,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BTextControl(BMessage *archive)
```

Initializes the BTextControl by assigning it a *label* and some *text*, both of which can be `NULL`. If the *label* is `NULL`, the entire bounds rectangle is assigned to the text. Otherwise, half the view is assigned to the label and half to the text, though the exact proportion can be changed by the `SetDivider()` function. The label always is on the left and the text always on the right. By default, both label and text are aligned at the left margins of their respective sections; call `SetAlignment()` to alter the alignment.

The *message* parameter is the same as the one declared for the BControl constructor. It establishes a model for the messages the BTextControl will send when it's invoked. It can be `NULL`. See `SetMessage()` and`SetTarget()` in the BInvoker class and `Invoke()` in the BControl class for more information.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change. When the BTextControl is attached to a window, it will be resized to the optimal height for displaying the label and text.

See also: `SetDivider()`, `SetAlignment()`, `BInvoker::SetMessage()`, `BInvoker::SetTarget()`, `BControl::Invoke()`

~BTextControl()

```
virtual ~BTextControl()
```

Frees memory allocated by the BTextControl and its BTextView child.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BTextControl object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BTextControl object, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, which will archive the child BTextView if the *deep* flag is `true`, then adds the alignment of the label and the text, the modification message, and the divider to the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of **AttachedToWindow()** to set up its child BTextView and to make the view and low colors of the BTextControl the same as the background view color of its new parent . This function also adjusts the height of the BTextControl to fit the height of the label and text, given the fonts used to display them.

See also: **BView::AttachedToWindow()**, **BControl::AttachedToWindow()**

Divider() see SetDivider()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the label. (The BTextControl defers to its child BTextView to draw the editable text string.)

See also: **BView::Draw()**

GetAlignment() see SetAlignment()

GetPreferredSize() , ResizeToPreferred()

virtual void **GetPreferredSize(**float **width*, float **height***)**

virtual void **ResizeToPreferred(**void**)**

GetPreferredSize() calculates the optimal size for the BTextControl to display the label and the text, given its current font and the current font of its BTextView child; it places the result in the variables that the *width* and *height* arguments refer to. **ResizeToPreferred()** resizes the BTextControl to its preferred size, keeping its left and top sides constant, and adjusts the size of the BTextView to fit.

See also: **BView::GetPreferredSize()**

MakeFocus()

virtual void **MakeFocus(**bool *flag* = true**)**

Passes the **MakeFocus()** instruction on to the child BTextView. If the *flag* is `true`, this function selects all the text in the child BTextView, which becomes the new focus view for the window. If the *flag* is `false`, the child will no longer be the focus view. If the text has changed when the child ceases to be the focus view, the BTextControl is considered to have been invoked; a copy of its model message is posted so that it will be delivered to the target handler.

Note that the BTextControl itself never becomes the focus view, so will return `false` to all **IsFocus()** queries.

See also: **BView::MakeFocus()**

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Handles scripting requests for the BTextControl.

See also: **BHandler::MessageReceived()**

ModificationMessage() see SetModificationMessage()

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Makes the child BTextView the focus view when the user clicks in the BTextControl but outside the text.

See also: **BTextView::MouseDown()**

ResizeToPreferred() see GetPreferredSize()

ResolveSpecifier()

virtual BHandler ***ResolveSpecifier(***BMessage **message*, int32 *index*, BMessage **specifier*, int32 *command*, const char **property***)**

Resolves specifiers for the "Text" and "Value" properties. See "Scripting Support" in the class overview and "Scripting" in **The Application Kit** chapter for more information.

See also: `BHandler::ResolveSpecifier()`

SetAlignment() , GetAlignment()

virtual void **SetAlignment(**alignment *forLabel*, alignment *forText***)**

void **GetAlignment(**alignment **forLabel*, alignment **forText***)** const

These functions set and report the alignment of the label and the text within their respective portions of the view. Three settings are possible:

B_ALIGN_LEFT	The label or text is aligned at the left boundary of its part of the view rectangle.
B_ALIGN_RIGH T	The label or text is aligned at the right boundary of its part of the view rectangle.
B_ALIGN_CEN TER	The label or text is centered within its part of the view rectangle.

The default alignment is `B_ALIGN_LEFT` for both label and text.

See also: `SetDivider()`

SetDivider() , Divider()

virtual void **SetDivider(**float *xCoordinate***)**

float **Divider(**void**)** const

These functions set and return the *x* coordinate value that marks the division between the label portion of the view rectangle on the left and the text portion on the right. It's stated in the coordinate system of the BTextControl.

See also: the BTextControl constructor

SetEnabled()

virtual void **SetEnabled(**bool *enabled***)**

Disables the BTextControl if the *enabled* flag is `false`, and reenables it if *enabled* is `true`. BTextControls are enabled by default.

This function augments the BControl version of **`SetEnabled()`**. When the control is disabled, it makes the text unselectable (and therefore uneditable) and draws it in a way that displays its disabled state. When the control is re-enabled, it makes the text editable (and therefore selectable) and draws it as normal text.

See also: **`BControl::SetEnabled()`**

SetModificationMessage() , ModificationMessage()

virtual void **SetModificationMessage(**BMessage ***message*)**
BMessage ***ModificationMessage(**void**)** const

These functions set and return the message that the BTextControl sends to its target when the user begins to enter or edit text.

`SetModificationMessage()` assigns *message* to the BTextControl, freeing the message previously assigned, if any. The message becomes the responsibility of the BTextControl object and will be freed only when it's replaced by another message or the BTextControl is freed; you shouldn't free it yourself. Passing a `NULL` pointer to this function deletes the current modification message without replacing it.

The assigned BMessage becomes the model for the message that the BTextControl sends whenever the user modifies the text when the child BTextView is the focus view.

Before sending the message, the BTextControl adds these two pieces of information to it:

Data name	Type code	Description
"when"	**`B_INT64_TYPE`**	When the user modified the text, as measured by the number of microseconds since 12:00:00 AM January 1, 1970..
"source"	**`B_POINTER_TYPE`**	A pointer to the BTextControl object.

These names should not be used for any data that you place in the model *message*.

`ModificationMessage()` returns the model message.

SetText() , Text()

virtual void **SetText(**const char ***text*)**
const char ***Text(**void**)** const

These functions set and return the text displayed by the BTextControl—or rather by its child BTextView. The text is first set by the constructor.

TextView()

BTextView ***TextView**(void) const

Returns the child BTextView object that handles the BTextControl's editing chores.

See also: the BTextView class

WindowActivated()

virtual void **WindowActivated**(bool *active*)

Makes sure that the BTextControl is redrawn when the window is activated and deactivated, if its child BTextView is the current focus view. This gives the BTextControl a chance to draw or erase the indication that it's the current focus for keyboard navigation.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_a_label"	`B_INT32_TYPE`	Alignment of label in frame.
"_a_text"	`B_INT32_TYPE`	Alignment of text in frame.
"_divide"	`B_FLOAT_TYPE`	x-coordinate dividing label and text in frame.
"_mod_msg"	`B_MESSAGE_TYPE`	Invocation message for the control (only present if one was defined).

The following view is added to the "_views" field (deep copy only):

Name	Level	Meaning
"_input_"	0	Text input view.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTextControl Function Summary

Function Summary

Function	Description
Archive()	Archives a BTextControl into a BMessage
AttachedToWindow()	Provided for implementation
Divider()	Returns the dividing point between the label and text
Draw()	Provided for implementation
GetAlignment()	Returns the alignment for the label and text
GetPreferredSize()	Provided for implementation
Instantiate()	Creates a BTextControl from a BMessage
MakeFocus()	Provided for implementation
MessageReceived()	Provided for implementation
ModificationMessage()	Returns the modification message for the control
MouseDown()	Provided for implementation
ResizeToPreferred()	Provided for implementation
ResolveSpecifier()	Provided for implementation
SetAlignment()	Sets the alignment for the label and text
SetDivider()	Sets the dividing point between the label and text
SetEnabled()	Provided for implementation

SetModificationMessage()	Sets the modification message for the control
SetText()	Sets the displayed text
Text()	Returns the displayed text
BTextControl()	Constructor
~BTextControl()	Destructor
TextView()	Returns the BTextView object controlling the editing
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScrollBar

Derived from: public BView
Declared in: be/interface/ScrollBar.h
Library: libbe.so
Summary: *more...*

A BScrollBar object displays a vertical or horizontal scroll bar that users can operate to scroll the contents of another view, a *target view*. Scroll bars usually are grouped as siblings of the target view under a common parent. That way, when the parent is resized, the target and scroll bars can be automatically resized to match. (A companion class, BScrollView, defines just such a container view; a BScrollView object sets up the scroll bars for a target view and makes itself the parent of the target and the scroll bars.)

The Update Mechanism

BScrollBars are different from other views in one important respect: All their drawing and event handling is carried out within the Application Server, not in the application. A BScrollBar object doesn't receive `Draw()` or `MouseDown()` notifications; the server intercepts updates and interface messages that would otherwise be reported to the BScrollBar and handles them itself. As the user moves the knob on a scroll bar or presses a scroll arrow, the Application Server continuously refreshes the scroll bar's image on-screen and informs the application with a steady stream of `B_VALUE_CHANGED` messages.

The window dispatches these messages by calling the BScrollBar's `ValueChanged()` function. Each function call notifies the BScrollBar of a change in its value and, consequently, of a need to scroll the target view.

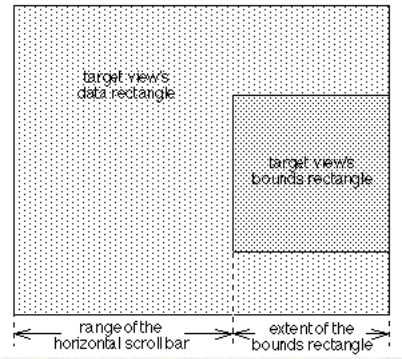
Confining the update mechanism for scroll bars to the Application Server limits the volume of communication between the application and server and enhances the efficiency of scrolling. The application's messages to the server can concentrate on updating the target view as its contents are being scrolled, rather than on updating the scroll bars themselves.

Value and Range

A scroll bar's value determines what the target view displays. The assumption is that the left coordinate value of the target view's bounds rectangle should match the value of the horizontal scroll bar, and the top of the target view's bounds rectangle should match the value of the vertical scroll bar. When a BScrollBar is notified of a change of value (through `ValueChanged()`), it calls the target view's `ScrollTo()` function to put the new value at the left or top of the bounds rectangle.

The value reported in a `ValueChanged()` notification and passed to `ScrollTo()` depends on where the user moves the scroll bar's knob and on the range of values the scroll bar represents. The range is first set in the BScrollBar constructor and can be modified by the `SetRange()` function.

The range must be large enough to bring all the coordinate values where the target view can draw into its bounds rectangle. If everything the target view can draw is conceived as being enclosed in a "data rectangle," the range of a horizontal scroll bar must extend from a minimum that makes the left side of the target's bounds rectangle coincide with the left side of its data rectangle, to a maximum that puts the right side of the bounds rectangle at the right side of the data rectangle. This is illustrated in part below:



As this illustration helps demonstrate, the maximum value of a horizontal scroll bar can be no less than the right coordinate value of the data rectangle minus the width of the bounds rectangle. Similarly, for a vertical scroll bar, the maximum value can be no less than the bottom coordinate of the data rectangle minus the height of the bounds rectangle. The range of a scroll bar subtracts the dimensions of the target's bounds rectangle from its data rectangle. (The minimum values of horizontal and vertical scroll bars can be no greater than the left and top sides of the data rectangle.)

What the target view can draw may change from time to time as the user adds or deletes data. As this happens, the range of the scroll bar should be updated with the **`SetRange()`** function. The range may also need to be recalculated when the target view is resized.

Coordination

Scroll bars control the target view, but a target can also be scrolled without the intervention of its scroll bars (by calling **`ScrollTo()`** or **`ScrollBy()`** directly). Therefore, not only must a scroll bar know about its target, but a target view must know about its scroll bars. When a BScrollBar sets its target, the target BView is notified and records the identity of the BScrollBar.

The two objects communicate whenever the display changes: When the scroll bar is the instrument that initiates scrolling, **`ValueChanged()`** calls the target view's **`ScrollTo()`** function. To cover cases of target-initiated scrolling, **`ScrollTo()`** calls the BScrollBar's **`SetValue()`** function so that the scroll bars can be updated on-screen. **`SetValue()`** in turn calls **`ValueChanged()`**, which makes sure the exchange of function calls doesn't get too circular.

Scroll Bar Options

Users have control over some aspects of how scroll bars look and behave. With the ScrollBar preferences application, they can choose:

- Whether the knob should be a fixed size, or whether it should grow and shrink to proportionally represent how much of a document (how much of the data rectangle) is visible within the target view. A proportional knob is the default.
- Whether double, bidirectional scroll arrows should appear on each end of the scroll bar, or whether each end should have only a single, unidirectional arrow. Double arrows are the default.
- Which of three patterns should appear on the knob.
- What the size of the knob should be—the minimum length of a proportional knob or the fixed length of a knob that's not proportional. The default length is 15 pixels.

When this class constructs a new BScrollBar, it conforms the object to the choices the user has made.

See also: `set_scroll_bar_info()`, **`BView::ScrollBar()`**, the BScrollView class

Hook Functions

ValueChanged()

Scrolls the target view when the BScrollBar is informed that its value has changed; can be augmented to coordinate other activities with the change in value..

Constructor and Destructor

BScrollBar()

```
BScrollBar(BRect frame, const char *name, BView *target,
    float min, float max, orientation posture)
BScrollBar(BMessage *archive)
```

Initializes the BScrollBar and connects it to the *target* view that it will scroll. It will be a horizontal scroll bar if *posture* is `B_HORIZONTAL` and a vertical scroll bar if *posture* is `B_VERTICAL`.

The range of values that the scroll bar can represent at the outset is set by *min* and *max*. These values should be calculated from the boundaries of a rectangle that encloses the entire contents of the target view—everything that it can draw. If *min* and *max* are both 0, the scroll bar is disabled and the knob is not drawn.

The object's initial value is 0 even if that falls outside the range set for the scroll bar.

The other arguments, *frame* and *name*, are the same as for other BViews:

- The *frame* rectangle locates the scroll bar within its parent view. For consistency in the user interface, a horizontal scroll bar should be `B_H_SCROLL_BAR_HEIGHT` coordinate units high, and a vertical scroll bar should be `B_V_SCROLL_BAR_WIDTH` units wide.
- The BScrollBar's *name* identifies it and permits it to be located by the `FindView()` function. It can be `NULL`.

Unlike other BViews, the BScrollBar constructor doesn't set an automatic resizing mode. By default, scroll bars have the resizing behavior that befits their posture—horizontal scroll bars resize themselves horizontally (as if they had a resizing mode that combined `B_FOLLOW_LEFT_RIGHT` with `B_FOLLOW_BOTTOM`) and vertical scroll bars resize themselves vertically (as if their resizing mode combined `B_FOLLOW_TOP_BOTTOM` with `B_FOLLOW_RIGHT`).

~BScrollBar()

```
virtual ~BScrollBar()
```

Disconnects the scroll bar from its target.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BScrollBar object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BScrollBar object, the return value will be `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, then adds the BScrollBar's range, orientation, current value and proportion, and the size of its big and little steps to the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Makes sure that the Application Server is cognizant of the BScrollBar's value, if a value was set before the object was attached to a window.

See also: **BHandler::AttachedToWindow()**

GetRange() see SetRange()

GetSteps() see SetSteps()

Orientation()

orientation **Orientation(**void**)** const

Returns **B_HORIZONTAL** if the object represents a horizontal scroll bar and **B_VERTICAL** if it represents a vertical scroll bar.

See also: the BScrollBar constructor

SetProportion() , Proportion()

void **SetProportion(**float *ratio***)**

float **Proportion(**void**)** const

These functions set and return a value between 0.0 and 1.0 that represents the proportion of the entire document that can be displayed within the target view—the ratio of the width (or height) of the target's bounds rectangle to the width (or height) of its data rectangle. This

ratio determines the size of a proportional scroll knob relative to the whole scroll bar. It's not adjusted to take into account the minimum size of the knob.

The proportion should be reset as the size of the data rectangle changes (as data is entered and removed from the document) and when the target view is resized.

SetRange() , GetRange()

void **SetRange(**float *min*, float *max***)**
void **GetRange(**float **min*, float **max***)** const

These functions modify and return the range of the scroll bar. `SetRange()` sets the minimum and maximum values of the scroll bar to *min* and *max*. `GetRange()` places the current minimum and maximum in the variables that *min* and *max* refer to.

If the scroll bar's current value falls outside the new range, it will be reset to the closest value—either *min* or *max*—within range. `ValueChanged()` is called to inform the BScrollBar of the change whether or not it's attached to a window.

If the BScrollBar is attached to a window, any change in its range will be immediately reflected on-screen. The knob will move to the appropriate position to reflect the current value.

Setting both the minimum and maximum to 0 disables the scroll bar. It will be drawn without a knob.

See also: the BScrollBar constructor

SetSteps() , GetSteps()

void **SetSteps(**float *smallStep*, float *bigStep***)**
void **GetSteps(**float **smallStep*, float **bigStep***)** const

`SetSteps()` sets how much a single user action should change the value of the scroll bar—and therefore how far the target view should scroll. `GetSteps()` provides the current settings.

When the user presses one of the scroll arrows at either end of the scroll bar, its value changes by a *smallStep* coordinate units. When the user clicks in the bar itself (other than on the knob), it changes by a *bigStep* units. For an application that displays text, the small step of a vertical scroll bar should be large enough to bring another line of text into view.

The default small step is 1.0, which should be too small for most purposes; the default large step is 10.0, which is also probably too small.

Although the step values are specified using type float, only integral values should be specified; otherwise, the scroll bar won't behave as expected.

Currently, a BScrollBar's steps can be successfully set only after it's attached to a window.

See also: `ValueChanged()`

SetTarget() , Target()

void **SetTarget(**BView *_view_**)** void **SetTarget(**const char *_name_**)**
BView ***Target(**void**)** const

These functions set and return the target of the BScrollBar, the view that the scroll bar scrolls. `SetTarget()` sets the target to *view*, or to the BView identified by *name*. `Target()` returns the current target view. The target can also be set when the BScrollBar is constructed.

`SetTarget()` can be called either before or after the BScrollBar is attached to a window. If the target is set by *name*, the named view must eventually be found within the same window as the scroll bar. Typically, the target and its scroll bars are children of a container view that serves to bind them together as a unit.

When the target is successfully set, a pointer to the BScrollBar object is passed to the target view. This lets the target update its scroll bars when its contents are scrolled.

See also: the BScrollBar constructor, `ValueChanged()`, **`BView::ScrollBar()`**

SetValue() , Value()

void **SetValue(**float _value_**)**
float **Value(**void**)** const

These functions modify and return the value of the scroll bar. The value is usually set as the result of user actions; `SetValue()` provides a way to do it programmatically. `Value()` returns the current value, whether set by `SetValue()` or by the user.

`SetValue()` assigns a new *value* to the scroll bar and calls the `ValueChanged()` hook function, whether or not the new value is really a change from the old. If the *value* passed lies outside the range of the scroll bar, the BScrollBar is reset to the closest value within range—that is, to either the minimum or the maximum value previously specified.

If the scroll bar is attached to a window, changing its value updates its on-screen display. The call to `ValueChanged()` enables the object to scroll the target view so that it too is updated to conform to the new value.

The initial value of a scroll bar is 0.

See also: `ValueChanged()`, `SetRange()`

Target() see SetTarget()

Value() see SetValue()

ValueChanged()

virtual void **ValueChanged(**float *newValue***)**

Responds to a notification that the value of the scroll bar has changed to *newValue*. For a horizontal scroll bar, this function interprets *newValue* as the coordinate value that should be at the left side of the target view's bounds rectangle. For a vertical scroll bar, it interprets *newValue* as the coordinate value that should be at the top of the rectangle. It calls **ScrollTo()** to scroll the target's contents into position, unless they have already been scrolled.

ValueChanged() is called as the result both of user actions (**B_VALUE_CHANGED** messages received from the Application Server) and of programmatic ones. Programmatically, scrolling can be initiated by the target view (calling **ScrollTo()**) or by the BScrollBar (calling **SetValue()** or **SetRange()**).

In all these cases, the target view and the scroll bars need to be kept in synch. This is done by a chain of function calls: **ValueChanged()** calls **ScrollTo()**, which in turn calls **SetValue()**, which then calls **ValueChanged()** again. It's up to **ValueChanged()** to get off this merry-go-round, which it does by checking the target view's bounds rectangle. If *newValue* already matches the left or top side of the bounds rectangle, if forgoes calling **ScrollTo()**.

ValueChanged() does nothing if a target BView hasn't been set—or if the target has been set by name, but the name doesn't correspond to an actual BView within the scroll bar's window.

Derived classes can override this function to interpret *newValue* differently, or to do something in addition to scrolling the target view.

See also: **SetTarget()**, **SetValue()**, **BView::ScrollTo()**

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_range" (array)	`B_FLOAT_TYPE`	Minimum and maximum values of the scroll bar.
"_steps" (array)	`B_FLOAT_TYPE`	Small and large steps of the scroll bar.
"_val"	`B_FLOAT_TYPE`	Current value of the scroll bar.
"_orient"	`B_INT32_TYPE`	Orientation of the scroll bar.
"_prop"	`B_FLOAT_TYPE`	Proportion of the document visible in the target view.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScrollBar Function Summary

Function Summary

Function	Description
Archive()	Archives a BScrollBar into a BMessage
AttachedToWindow()	Provided for implementation
GetRange()	Returns the numerical limits of the scroll bar
GetSteps()	Returns the step amounts of the scroll bar
Instantiate()	Creates a BScrollBar from a BMessage
Orientation()	Returns orientation of scrollbar (vertical or horizontal)
Proportion()	Returns the proportion of the document viewable within the target view
BScrollBar()	Constructor
~BScrollBar()	Destructor
SetProportion()	Sets the proportion of the document viewable within the target view
SetRange()	Sets the numerical limits of the scroll bar
SetSteps()	Sets the step amounts of the scroll bar
SetTarget()	Sets the target view of the scroll bar
SetValue()	Sets the value of the scroll bar
Target()	Returns the target view of the scroll bar

Value()	Returns the value of the scroll bar
ValueChanged()	Hook called when the value of the scroll bar changes

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScrollView

Derived from: public BView
Declared in: be/interface/ScrollView.h
Library: libbe.so
Summary: *more...*

A BScrollView object is a container for some other view (the *target view*). The BScrollView creates and positions scroll bars around the target view, and makes itself the parent of the scroll bars and the target view. When the user moves the scroll bars, the target view scrolls.

Constructor and Destructor

BScrollView()

```
BScrollView(const char *name, BView *target,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = 0,
    bool horizontal = false,
    bool vertical = false,
    border_style border = B_FANCY_BORDER)
BScrollView(BMessage *archive)
```

Creates a BScrollView that has *target* as its target view. If *horizontal* is `true`, a horizontal scroll bar is added to the bottom of the target view. If *vertical* is `true`, a vertical scroll bar is added to the target's right side. The BScrollView's frame is manufactured for you to contain the target view, the scroll bars, and the optional border; the location of the target view doesn't change. The scroll bar "thickness" constants, defined in **`ScrollBar.h`**, will help you compute the size of the frame:

`B_V_SCROLL_BAR_WIDTH`	The width of a vertical scroll bar.
`B_H_SCROLL_BAR_HEIGHT`	The height of a horizontal scroll bar.

The target and the scroll bars are added as (sibling) children of the scroll view, and the target's **`TargetedByScrollView()`** function is called. The target must not already be the child of some other view.

The *border* argument can be one of three values:

`B_PLAIN_BORDER`	Draw a border consisting of just a simple line around the target view and scroll bars.
`B_FANCY_BORDER`	Draw a fancier border that looks like a 3D groove inset into the surface of the view.
`B_NO_BORDER`	Don't draw a border.

If the resize mode of the target view is **`B_FOLLOW_ALL_SIDES`**, it and the scroll bars will be automatically resized to fill the scroll view whenever the scroll view is resized.

The scroll bars created by the BScrollView have an initial range of 0 to 1000. Use the **`ScrollBar()`** function to retrieve the scroll bars and reset their ranges.

The *name*, *resizingMode*, and *flags* arguments are identical to those declared in the BView class and are passed to the BView constructor. If a border is requested, **`B_WILL_DRAW`** is automatically added to the *flags* mask. The other two arguments are passed to the BView class unchanged.

See also: the BView constructor, **`BView::TargetedByScrollView()`**

~BScrollView()

virtual **~BScrollView()**

Does nothing.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BScrollView object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the message doesn't contain data for an archived BScrollView object, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

Archive()

virtual status_t **Archive(**BMessage ***archive**, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, which will archive the target view and scroll bars if the *deep* flag is `true`. This function then adds the BScrollView's border style to the *archive* message.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Resizes scroll bars belonging to BScrollViews that occupy the right bottom corner of a document window (**B_DOCUMENT_WINDOW**) so that room is left for the resize knob. This function assumes that vertical scroll bars are **B_V_SCOLL_BAR_WIDTH** units wide and horizontal scroll bars are **B_H_SCROLL_BAR_HEIGHT** units high. It doesn't check to make sure the window is actually resizable.

This function also sets the default high color to a medium shade of gray.

See also: **BView::AttachedToWindow()**

Border() see SetBorder()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the border around the target view and scroll views, provided a border was requested when the BScrollView was constructed.

See also: the BScrollView constructor, **BView::Draw()**

IsBorderHighlighted() see SetBorderHighlighted()

ScrollBar()

BScrollBar ***ScrollBar(**orientation *posture***)** const

Returns the horizontal scroll bar if *posture* is **B_HORIZONTAL** and the vertical scroll bar if *posture* is **B_VERTICAL**. If the BScrollView doesn't contain a scroll bar with the requested orientation, this function returns NULL.

See also: the BScrollBar class

SetBorder() , Border()

virtual void **SetBorder(**border_style *border***)**
border_style **Border(**void**)** const

These functions set and return the style of the BScrollView's border, which may be **B_PLAIN_BORDER**, **B_FANCY_BORDER**, or **B_NO_BORDER**. The border is originally set by the constructor. The three constants and the border's effect on the BScrollView are discussed there.

See also: the BScrollView constructor

SetBorderHighlighted() , IsBorderHighlighted()

virtual status_t **SetBorderHighlighted(**bool *highlighted***)**
bool **IsBorderHighlighted(**void**)** const

SetBorderHighlighted() highlights the border of the BScrollView when the *highlighted* flag is true, and removes the highlighting when the flag is false. This function works by calling **Draw()**. However, it works only for a border in the **B_FANCY_BORDER** style. If successful, it returns **B_OK**. Otherwise, it returns **B_ERROR**.

IsBorderHighlighted() returns whether the border is currently highlighted. The return value is always false for a BScrollView that doesn't have a border or has only a "plain" one.

Highlighting a BScrollView's border shows that the target view is the current focus view for the window. Typically, the target view calls **SetBorderHighlighted()** from its **MakeFocus()** function when the focus changes. (The target knows that it's inside a BScrollView because of the **TargetedByScrollView()** notification it received.)

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_style"	`B_INT32_TYPE`	Border type.

The following views are added to the "_views" field (deep copy only):

Name	Level	Meaning
target	0	Target view for the object.
"_VSB_"	0	Vertical BScrollBar.
"_HSB_"	0	Horizontal BScrollBar.

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the border type is `B_FANCY_BORDER`, the "_style" field won't be found in the archive.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScrollView Function Summary

Function Summary

Function	Description
Archive()	Archives a BScrollView into a BMessage
AttachedToWindow()	Provided for implementation
Border()	Returns the border style
Draw()	Provided for implementation
Instantiate()	Creates a BScrollView from a BMessage
IsBorderHighlighted()	Tests whether border highlights are enabled
ScrollBar()	Returns the scroll bar with a given orientation
BScrollView()	Constructor
~BScrollView()	Destructor
SetBorder()	Sets the border style
SetBorderHighlighted()	Enables border highlights

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

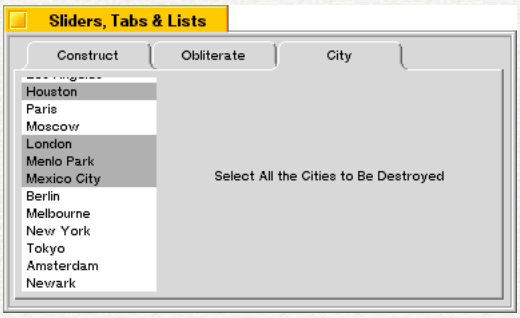
Copyright © 2000 Be, Inc. All rights reserved..

BListView

Derived from: public BView, public BInvoker
Declared in: be/interface/ListView.h
Library: libbe.so
Summary: *more...*

A BListView displays a list of items the user can select and invoke. Each item is a kind of BListItem object. The BListView manages the layout of the list and the interaction with the user; it leaves the display of each item to the BListItem objects. A BListItem is not a view and draws only when called upon by the BListView.

The figure below depicts a window that contains a sample BListView.



A BListView displays all items unindented at a single level. A derived class, BOutlineListView, can arrange items in an hierarchical outline.

Lists and List Views

This class is based on the BList class of the Support Kit. It implements counterparts for *all* BList member functions, so you can treat a BListView object just like a BList. BListView simply makes the list visible.

BListView functions work identically to their BList counterparts, except for two things:

- A BListView assumes that the list contains pointers to BListItem objects, not `void*` pointers.
- A BListView makes sure that the on-screen display is properly updated whenever the items in the list change.

In both classes, the list keeps track of data pointers—`void*` pointers in the case of BList and pointers to BListItems in the case of BListView. Adding an item to the list adds only the pointer; the pointed-to object isn't copied.

Updating the List

When the contents of the list change, the BListView makes sure the visible list on-screen is updated. However, since it records only pointers to data, it can know that something changed only when a BListItem is added or removed. If an item pointer remains the same but the data the item displays is altered, the BListView won't know about it. In this case, you must force the list to be redrawn (by calling the **`InvalidateItem()`** function or BView's **`Invalidate()`**).

Selecting and Invoking Items

The user can click an item in the list to select it and double-click an item to both select and invoke it. The user can also select and invoke items from the keyboard. The navigation keys (such as Down Arrow, Home, and Page Up) select items; Enter and the space bar invoke them.

By default, a BListView permits only one item to be selected at a time. However, at construction and with the **`SetListType()`** function, you can set it up to allow multiple selections. The user can make contiguous extensions to the current selection by holding down a Shift key, and discontinuous extensions by holding down an Option key.

The BListView highlights items as they're selected, but otherwise it doesn't define what, if anything, should take place when the selection changes. You can determine that yourself either by implementing a **`SelectionChanged()`** function in a derived class or by registering a *selection message* (a BMessage object) with the BListView. The function is called and the message is delivered to a target destination whenever the user modifies the selection.

Similarly, the BListView doesn't define what it means to "invoke" an item. You can register a separate *invocation message* that's sent whenever the user double-clicks an item or presses Enter or the space bar while items are selected. For example, if the user double-clicks an item in a list of file names, the message might tell the BApplication object to open that file.

A BListView doesn't have default selection and invocation messages. Messages are sent only if registered with the **`SetSelectionMessage()`** and **`SetInvocationMessage()`** functions. Before sending either type of message, the BListView adds information to it identifying itself and the items that are currently selected. See the **`Invoke()`** function for details.

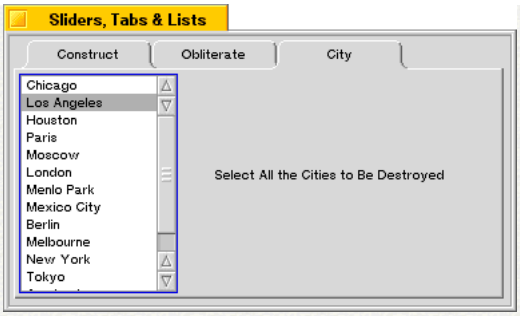
Autoscrolling and Dragging Items

BListView provides an autoscroll feature: There's an invisible area about 25 pixels high both above and below the list view; if the user clicks inside the list view and then drags into this autoscroll area, the contents of the list will scroll in the direction of the mouse.

BListView also provides optional item-dragging that let's the user reorder the items in the list. To turn on this feature, you have to subclass BListView and implement **`InitiateDrag()`** to return **`true`**.

Adding a Scroll Bar to a BListView

Note that although a BListView is scrollable, it doesn't provide scroll bars. You can place the BListView inside a BScrollView to get the more traditional appearance for your list:



The code for this might look something like this:

```
BListView *list = new BListView(r, "City", B_MULTIPLE_SELECTION_LIST);
list->AddItem(new BStringItem("Chicago"));
...
theView->AddChild(new BScrollView("scroll_cities", list,
        B_FOLLOW_LEFT | B_FOLLOW_TOP, 0, false, true));
```

See the BScrollView class for a more in-depth explanation of scrolling views.

See also: the BList class in the Support Kit, the BOutlineListView and BListItem classes

Hook Functions

`InitiateDrag()`
Can be implemented to permit users to drag items—for example, to reorder items in the list.

`SelectionChanged()`
Can be implemented to take collateral action each time the selection changes.

Constructor and Destructor

BListView()

```
BListView(BRect frame, const char *name,
    list_view_type type = B_SINGLE_SELECTION_LIST,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP, uint32 flags = B_WILL_DRAW |
B_NAVIGABLE | B_FRAME_EVENTS)
BListView(BMessage *archive)
```

Initializes the new BListView. The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The list *type* can be either:

B_SINGLE_SELECTION_LIST	The user can select only one item in the list at a time. This is the default setting.
B_MULTIPLE_SELECTION_LIST	The user can select any number of items by holding down an Option key (for discontinuous selections) or a Shift key (for contiguous selections).

The list begins life empty. Call **AddItem()** or **AddList()** to put items in the list. Call **Select()** to select one of the items so that it's highlighted when the list is initially displayed to the user.

See also: the BView constructor, **AddItem()**

~BListView()

```
virtual ~BListView()
```

Frees the selection and invocation messages, if any, and any memory allocated to hold the list of items, but not the items themselves.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BListView object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BListView object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

AddItem()

virtual bool **AddItem(**BListItem **item*, int32 *index***)**
virtual bool **AddItem(**BListItem **item***)**

Adds an *item* to the BListView at *index*—or, if no index is supplied, at the end of the list. If necessary, additional memory is allocated to accommodate the new item.

Adding an item never removes an item already in the list. If the item is added at an index that's already occupied, items currently in the list are bumped down one slot to make room.

If *index* is out of range (greater than the current item count, or less than zero), this function fails and returns **false**. Otherwise, it returns **true**.

See also: **BList::AddItem()**

AddList()

virtual bool **AddList(**BList **list*, int32 *index***)**
virtual bool **AddList(**BList **list***)**

Adds the contents of another *list* to this BListView. The items from the BList are inserted at *index*—or, if no index is given, they're appended to the end of the list. If the index is out of range, the function fails and returns **false**. If successful, it returns **true**.

The BListView doesn't check to be sure that all the items it adds from the *list* are pointers to BListItem objects. It assumes that they are; if the assumption is false, subsequent BListView operations will fail.

See also: **AddItem()**, **BList::AddList()**

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets up the BListView and makes the BWindow to which it has become attached the target for the messages it sends when items are selected or invoked—provided another target hasn't already been set. In addition, this function calls **Update()** for each item in the list to give it a chance to adjust its layout. The BListView's vertical scroll bar is also adjusted.

This function is called for you when the BListView becomes part of a window's view hierarchy.

See also: **BView::AttachedToWindow()**, **BInvoker::SetTarget()**, **BListItem::Update()**

CountItems()

int32 **CountItems**(void) const

Returns the number of BListItems currently in the list.

See also: BList::CountItems()

CurrentSelection()

int32 **CurrentSelection**(int32 *index* = 0) const

Returns the index of a currently selected item in the list, or a negative number if no item is selected. The domain of the *index* passed as an argument is the current set of selected items; the first selected item is at *index* 0, the second at *index* 1, and so on, even if the selection is not contiguous. The domain of the returned index is the set of all items in the list.

To get all currently selected items, increment the passed index until the function returns a negative number:

```
BListItem *item;
int32 selected;
while ( (selected = myListView->CurrentSelection(i)) >= 0 ) {
   item = ItemAt(selected);
   . . .
}
```

See also: Select()

Deselect() , DeselectAll() , DeselectExcept()

void **Deselect**(int32 *index*)

void **DeselectAll**(void)

void **DeselectExcept**(int32 *start*, int32 *finish*)

These functions deselect the item at *index*, all the items, or all the items except those from index *start* through index *finish*.

See also: Select()

DoForEach()

void **DoForEach**(bool (**func*)(BListItem *))

void **DoForEach(**bool (**func*)(BListItem *, void *), void **arg2***)**

Calls the *func* function once for each item in the BListView. BListItems are visited in order, beginning with the first one in the list (index 0) and ending with the last. If a call to *func* returns **true**, the iteration is stopped, even if some items have not yet been visited.

func must be a function that takes one or two arguments. The first argument is a pointer to the BListItem; the second argument, if *func* requires one, is passed to **DoForEach()** as *arg2*.

See also: **BList::DoForEach()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Calls upon every item in the *updateRect* area of the view to draw itself.

Draw() is called for you whenever the list view is to be updated or redisplayed; you don't need to call it yourself. You also don't need to reimplement it; to change the way items are drawn, define a new version of **DrawItem()** in a class derived from BListItem.

See also: **BView::Draw()**, **BListItem::DrawItem()**

FirstItem() see ItemAt()

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Updates the on-screen display in response to a notification that the BListView's frame rectangle has been resized. In particular, this function looks for a vertical scroll bar that's a sibling of the BListView. It adjusts this scroll bar to reflect the way the list view was resized, under the assumption that it must have the BListView as its target.

FrameResized() is called automatically at the appropriate times; you shouldn't call it yourself.

See also: **BView::FrameResized()**

HasItem()

bool **HasItem(**BListItem **item***)** const

Returns **true** if *item* is in the list, and **false** if not.

See also: **BList::HasItem()**

IndexOf()

int32 **IndexOf(**BListItem **item***)** const
int32 **IndexOf(**BPoint **point***)** const

Returns the index where a particular *item*—or the item whose display rectangle includes a particular *point*—is located in the list. If the *item* is in the list more than once, the index returned will be the position of its first occurrence. To determine whether an item lies at the specified *point*, only the *y*-coordinate value of the point is considered.

If the *item* isn't in the list or the *y*-coordinate of the *point* doesn't intersect with the data rectangle of the BListView, the return value will be a negative number.

See also: `BList::IndexOf()`

InitiateDrag()

virtual bool **InitiateDrag(**BPoint *point*, int32 *index*, bool *wasSelected***)**

Implemented by derived classes to permit users to drag items. This function is called from the BListView's `MouseDown()` function; it should initiate the drag-and-drop operation and return `true`, or refuse to do so and return `false`. By default, it always returns `false`.

The *point* that's passed to `InitiateDrag()` is the same as the point passed to `MouseDown()`; it's where the cursor was located when the user pressed the mouse button. The *index* of the item under the cursor (the item that would be dragged) is passed as the second argument, and the *wasSelected* flag indicates whether or not the item was selected before the mouse button went down.

A BListView allows users to autoscroll the list by holding the mouse button down and dragging outside its frame rectangle. If a derived class implements `InitiateDrag()` to drag an item each time the user moves the mouse with a button down, it will hide this autoscrolling behavior. Therefore, derived classes typically permit users to drag items only if they're already selected (if *wasSelected* is `true`). In other words, it takes two mouse-down events to drag an item—one to select it and one to begin dragging it.

See also: `BView::DragMessage()`

InvalidateItem()

void **InvalidateItem(**int32 *index***)**

Invalidates the item at *index* so that an update message will be sent forcing the BListView to redraw it.

See also: `BView::Invalidate()`

Invoke()

virtual status_t **Invoke(**BMessage **message* = NULL**)** const

Augments the BInvoker version of **Invoke()** to add three pieces of information to each message the BListView sends:

Data name	Type code	Description
"when"	**B_INT64_TYPE**	When the message is sent, as measured by the number of microseconds since 12:00:00 AM 1970.
"source"	**B_POINTER_TYPE**	A pointer to the BListView object.
"index"	**B_INT32_TYPE**	An array containing the index of every selected item.

This function is called to send both the selection message and the invocation message. It can also be called from application code. The default target of the message (established by **AttachedToWindow()**) is the BWindow where the BListView is located.

What it means to "invoke" selected items depends entirely on the invocation BMessage and the receiver's response to it. This function does nothing but send the message.

See also: **Select()**, **SetInvocationMessage()**, **BInvoker::SetTarget()**

IsEmpty() see MakeEmpty()

IsItemSelected()

bool **IsItemSelected(**int32 *index***)** const

Returns `true` if the item at *index* is currently selected, and `false` if it's not.

See also: **CurrentSelection()**

ItemAt() , FirstItem() , LastItem()

BListItem ***ItemAt(**int32 *index***)** const

BListItem ***FirstItem(**void**)** const

BListItem ***LastItem(**void**)** const

The first of these functions returns the BListItem at *index*, or **NULL** if the index is out of range. The other two functions return the very first and very last items in the list, or `NULL` if the list is empty. None of the functions alters the contents of the list—they don't remove the returned item.

See also: `Items()`, `BList::FirstItem()`, `BList::LastItem()`, `BList::ItemAt()`

ItemFrame()

```
BRect ItemFrame(int32 index)
```

Returns the frame rectangle of the BListItem at *index*. The rectangle is stated in the coordinate system of the BListView and defines the area where the item is drawn. Items can differ in height, but all have the same width.

See also: `DrawItem()`

Items()

```
const BListItem **Items(void) const
```

Returns a pointer to the BListView's list of BListItems. You can index directly into the list of items if you're certain that the index is in range:

```
BListItem *item = Items()[index];
```

Although the practice is discouraged, you can also step through the list of items by incrementing the list pointer that `Items()` returns. Be aware that the list isn't null-terminated—you have to detect the end of the list by some other means. The simplest method is to count items:

```
BListItem **ptr = myListView->Items();

for ( long i = myListView->CountItems(); i > 0; i-- )
{
   . . .
   *ptr++;
}
```

You should never use the items pointer to alter the contents of the list.

See also: `DoForEach()`, `SortItems()`, `BList::Items()`

KeyDown()

```
virtual void KeyDown(const char *bytes, int32 numBytes)
```

Permits the user to operate the list using the following keys:

Keys	Perform Action

Up Arrow and Down Arrow	Select the items that are immediately before and immediately after the currently selected item.
Page Up and Page Down	Select the items that are one viewful above and below the currently selected item—or the first and last items if there's no item a viewful away.
Home and End	Select the first and last items in the list.
Enter and the space bar	Invoke the current selection.

This function also incorporates the inherited BView version so that the Tab key can navigate to another view.

`KeyDown()` is called to report `B_KEY_DOWN` messages when the BListView is the focus view of the active window; you shouldn't call it yourself.

See also: **`BView::KeyDown()`**, `Select()`, `Invoke()`

LastItem() see ItemAt()

MakeEmpty() , IsEmpty()

virtual void **`MakeEmpty(`**void**`)`**
bool **`IsEmpty(`**void**`)`** const

`MakeEmpty()` empties the BListView of all its items, without freeing the BListItem objects.

`IsEmpty()` returns **`true`** if the list is empty (if it contains no items), and **`false`** otherwise.

See also: `RemoveItem()`, **`BList::MakeEmpty()`**

MakeFocus()

virtual void **`MakeFocus(`**bool *focused* = true**`)`**

Overrides the BView version of `MakeFocus()` to draw an indication that the BListView has become the focus for keyboard events when the *focused* flag is `true`, and to remove that indication when the flag is `false`.

See also: **`BView::MakeFocus()`**

MouseDown()

virtual void **`MouseDown(`**BPoint *point***`)`**

Responds to **B_MOUSE_DOWN** messages by selecting items, invoking them (if the mouse-down event is the second of a double-click), and autoscrolling the list (when the user drags with a mouse button down). This function also calls **InitiateDrag()** to give derived classes the opportunity to drag items. You can implement that function; you shouldn't override (or call) this one.

See also: **BView::MouseDown()**, **Select()**, **Invoke()**, **InitiateDrag()**

MoveItem() , ReplaceItem() , SwapItems()

bool **MoveItem(**int32 *from*, int32 *to***)**
bool **ReplaceItem(**int32 *index*, BListItem ***item***)**
bool **SwapItems(**int32 *index1*, int32 *index2***)**

MoveItem() moves the item located at the index *from* to the index specified by the *to* argument.

ReplaceItem() replaces the item located at the specified *index* with the specified *item*.

SwapItems() swaps the items specified by *index1* and *index2* in the list.

These functions return **true** if the requested operation is completed successfully, or **false** if the operation failed (for example, if a specified index is out of range).

RemoveItem() , RemoveItems()

virtual BListItem ***RemoveItem(**int32 *index***)** virtual bool **RemoveItem(**BListItem ***item***)**
virtual bool **RemoveItems(**int32 *index*, int32 *count***)**

RemoveItem() removes a single item from the BListView. If passed an *index*, it removes the item at that index and returns it. If there's no item at the index, it returns **NULL**. If passed an *item*, this function looks for that particular item in the list, removes it, and returns **true**. If it can't find the item, it returns **false**. If the item is in the list more than once, this function removes only its first occurrence.

RemoveItems() removes *count* number of items from the BListView, beginning with the item at *index*. If the list doesn't contain *count* items between *index* and the end of the list, this function removes as many items as there are.

The list is compacted after an item is removed. Therefore, you shouldn't try to empty a list (or a range within a list) by removing items incrementing indices. You should either start with the highest index and move towards the head of the list, or remove at the same index (the lowest in the range) some number of times. As an example of the latter, the following code removes the first five items in the list:

```
for ( int32 i = 0; i <= 4; i++ )
```

```
myListView->RemoveItem(0);
```

See also: `MakeEmpty()`, `BList::RemoveItem()`

ScrollTo()

virtual void **ScrollTo(**BPoint *point***)**
inline void **ScrollTo(**float *x*, float *y***)**

Augments the BView version of `ScrollTo()` to do some class-internal housekeeping when the list is scrolled. For all practical purposes, these functions are identical to their BView counterparts.

See also: `BView::ScrollTo()`

ScrollToSelection()

void **ScrollToSelection(**void**)**

Scrolls the list so that the first item in the current selection is visible.

Select()

void **Select(**int32 *index*, bool *extend* = false**)**
void **Select(**int32 *start*, int32 *finish*, bool *extend* = false**)**

Selects the item located at *index*—or all the items from the *start* index through the *finish* index—provided that none of the indices are out of range. If the *extend* flag is `false`, as it is by default, this function removes the highlighting from the previously selected item(s) and highlights the new selection. However, if the *extend* flag is `true`, the newly selected items are added to the current selection.

If you want your newly-selected item to be scrolled into view, you must call `ScrollToSelection()` after invoking `Select()`. The `Select()` function itself does not do any scrolling.

`Select()` can be called to set an initial selection in the list or change the current selection. It permits the program to select a number of items, even for a `B_SINGLE_SELECTION_LIST` list.

If you want to be informed of successful selections, you either have to implement `SelectionChanged()` or supply a selection message (through `SetSelectionMessage()`). The former is invoked and the latter is sent when an item is selected.

See also: `SetSelectionMessage()`, `Invoke()`, `SelectionChanged()` , `Deselect()`

SelectionChanged()

virtual void **SelectionChanged**(void)

Implemented by derived classes to do whatever they please when the selection changes.

SetInvocationMessage() , InvocationMessage() , InvocationCommand()

virtual void **SetInvocationMessage**(BMessage **message*)
BMessage ***InvocationMessage**(void) const
uint32 **InvocationCommand**(void) const

These functions set and return information about the BMessage that the BListView sends when currently selected items are invoked.

SetInvocationMessage() assigns *message* to the BListView, freeing any message previously assigned. The message becomes the responsibility of the BListView object and will be freed only when it's replaced by another message or the BListView is freed; you shouldn't free it yourself. Passing a `NULL` pointer to this function deletes the current message without replacing it.

When sending the message, the **Invoke()** function makes a copy of it and adds two pieces of relevant information—"when" the message is sent and the "source" BListView. These names should not be used for any data that you add to the invocation *message*.

InvocationMessage() returns a pointer to the BMessage and **InvocationCommand()** returns its `what` data member. The message belongs to the BListView; it can be altered by adding or removing data, but it shouldn't be deleted. To get rid of the current message, pass a `NULL` pointer to **SetInvocationMessage()**.

See also: **Invoke()**, the BMessage class

SetListType() , ListType()

virtual void **SetListType**(list_view_type *type*)
list_view_type **ListType**(void) const

These functions set and return the list type—whether or not it permits multiple selections. The **list_view_type** must be either **B_SINGLE_SELECTION_LIST** or **B_MULTIPLE_SELECTION_LIST**. The type is first set when the BListView is constructed.

See also: the BListView constructor

SetSelectionMessage() , SelectionMessage() , SelectionCommand()

virtual void **SetSelectionMessage(**BMessage **message***)**
BMessage ***SelectionMessage(**void**)** const
uint32 **SelectionCommand(**void**)** const

These functions set, and return information about, the message that a BListView sends whenever a new item is selected. They're exact counterparts to the functions described above under **SetInvocationMessage()**, except that the selection message is sent whenever an item in the list is selected, rather than when invoked. It's more common to take action (to initiate a message) when invoking an item than when selecting one.

See also: **Select()**, **SetInvocationMessage()**, **Invoke()**, the BMessage class

---

SortItems()

void ***SortItems(**int (**compareFunc*)(const void *, const void *)**)**

Rearranges the items in the list. The items are sorted using the *compareFunc* comparison function passed as an argument. This function should return a negative number if the first item is ordered before the second, a positive number if the second comes before the first, and 0 if the two items are ordered equivalently.

Although the comparison function is, in the usual manner for such functions, declared to take two `void*` arguments, each argument should be a pointer to an item in the list—in other words, a pointer to a BListItem pointer:

```
int compare_func(BListItem **firstArg, BListItem **secondArg);
```

See also: **Items()**, **BList::SortItems()**

TargetedByScrollView()

virtual void **TargetedByScrollView(**BScrollView **view***)**

Notes the fact that the BListView is the target of a BScrollView and arranges for the border of the BScrollView to be highlighted when the BListView is the current focus view of the active window.

See also: **BScrollView::SetBorderHighlighted()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Makes sure that the BScrollView that targets the BListView is redrawn when the window is activated and deactivated, if the BListView is the current focus view. This gives the BScrollView a chance to highlight or unhighlight its border, as appropriate.

See also: `BView::WindowActivated()`

Scripting Support

The BListView class implements the suite called "suite/vnd.Be-list-view" consisting of the following messages:

The Item Property

Message	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Returns the number of BListItems currently in the list.
B_EXECUTE_PROPERTY	B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER, B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Select and invoke the specified items, first removing any existing selection.

Individual BListItems in the list The "Item" property refers to individual BListItems in the list. The Selection Property	Message	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Returns int32 count of items in the selection.	
B_EXECUTE_PROPERTY	B_DIRECT_SPECIFIER	Invoke items in selection.	
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns int32 indices of all items in the selection.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER, B_RANGE_SPECIFIER, B_REVERSE_RANGE_SPECIFIER	Extends current selection or deselects specified items. Boolean field "data" chooses selection or deselection.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Select or deselect all items in the selection. Boolean field "data" chooses selection or deselection.	

The set of selected BListItems in the list The "Selection" property refers to the set of selected BListItems. If the BListView is of type B_SINGLE_SELECTION_LIST, then only one item can be	Field	Type code	Meaning

selected at any given instant; the list type cannot be overridden through the scripting interface. Attempting to access multiple selected list elements in a `B_SINGLE_SELECTION_LIST` results in an error.

Archived Fields **The Archive() function adds the following fields to its BMessage argument:**			
"_lv_type"	B_INT32_TYPE	The list type.	
"_l_items" (array)	B_MESSAGE_TYPE	List items in the list (deep copy only)	
"_msg"	B_MESSAGE_TYPE	Invocation message.	
"_2nd_msg"	B_MESSAGE_TYPE	Selection message.	

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if there is no selection message, the "_2nd_msg" field won't be found in the archive.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BListView Function Summary

Function Summary

Function	Description
AddItem()	Adds BListItem to the list
AddList()	Inserts contents of a BList of BListView to the list
AttachedToWindow()	Provided for implementation
CountItems()	Returns number of BListItems in the list
CurrentSelection()	Returns index of currently selected item in the list
DeselectAll()	Deselects all items in the list
Deselect()	Deselects an item in the list
DeselectExcept()	Deselects all list items except those in a given range
DoForEach()	Calls a function on each item in the list
Draw()	Provided for implementation
FirstItem()	Returns first item in the list
FrameResized()	Provided for implementation
HasItem()	Tests whether a given BListItem is in the list
IndexOf()	Locates or identifies an item in the list
InitiateDrag()	Hook to permit users to drag items
Instantiate()	Creates a BListView from a BMessage

InvalidateItem()	Invalidates the drawing rectangle of a list item
InvocationCommand()	Sets what field of invocation BMessage
InvocationMessage()	Returns BMessage sent to indicate invocation of item
Invoke()	Provided for implementation
IsEmpty()	Returns true if list is empty
IsItemSelected()	Tests whether the specified item is selected
ItemAt()	Returns item at given index
ItemFrame()	Returns frame of item at given index
Items()	Returns list of items in the list
KeyDown()	Provided for implementation
LastItem()	Returns last item in the list
ListType()	Returns list type
BListView()	Constructor
~BListView()	Destructor
MakeEmpty()	Empties the list
MakeFocus()	Provided for implementation
MouseDown()	Provided for implementation
MoveItem()	Change an item's position in the list
RemoveItem()	Removes an item from the list
RemoveItems()	Removes a run of items from the list

ReplaceItem()	Replaces an item in the list
ScrollTo()	Provided for implementation
ScrollToSelection()	Scrolls list so the first item of the selection is visible
Select()	Selects specified item or range of items
SelectionChanged()	Hook called each time the selection changes
SelectionCommand()	Returns what field of selection BMessage
SelectionMessage()	Returns BMessage sent to indicate selection of item
SetInvocationMessage()	Sets BMessage sent to indicate invocation of item
SetListType()	Sets list type
SetSelectionMessage()	Sets BMessage sent to indicate selection of item
SortItems()	Sorts list items with a user-specified ordering function
SwapItems()	Swaps the position of two items in the list
TargetedByScrollView()	Provided for implementation
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BListItem

Derived from: none
Mix-in classes: BArchivable
Declared in: be/interface/ListItem.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BListItem represents a single item in a BListView (or BOutlineListView). The BListItem object provides drawing instructions that can draw the item (through `DrawItem()`), and keeps track of the item's state. To use a BListItem, you must add it to a BListView through `BListView::AddItem()` (BOutlineListView provides additional item-adding functions). The BListView object provides the drawing context for the BListItem's `DrawItem()` function.

BListItem is abstract; each subclass must implement `DrawItem()`. BStringItem—the only BListItem subclass provided by Be—implements the function to draw the item as a line of text. For an example of a custom BListView subclass, see "BListItem: Creating a Custom List Item".

Synchronizing a List Item with its List View

A BListItem object doesn't automatically get redrawn when the item changes. If you change a list item's content or state, you must tell the item's owner (the BListView object) to redraw the item by calling **`BListView::InvalidateItem()`**. For example:

```
/* listItem belongs to listView. We change the state of the item... */
listItem->SetEnabled(false);

/* ...so we have to tell the list view to redraw it. */
listView->LockLooper();
listView->InvalidateItem(listView->IndexOf(listItem));
listView->UnlockLooper();
```

If you're making a lot of changes, you can flush them all at the same time through a single **`BListView::Invalidate()`** call:

```
listItemA->SetEnabled(false);
listItemB->SetEnabled(false);
listItemC->SetEnabled(false);
listView->LockLooper();
listView->Invalidate();
listView->UnlockLooper();
```

Note that you don't have to lock the list view's window to change one of its items—you only have to lock the window when you talk to the list view itself.

BListView provides its own smart version

Hook Functions

- `DrawItem()`
- `Update()`

Constructor and Destructor

BListItem() , ~BListItem()

```
BListItem(uint32 level = 0, bool expanded = true)
BListItem(BMessage *archive)
```

The constructors create a new BListItem object. The *level* and *expanded* arguments are only used if the item is added to a BOutlineListView object; see **BOutlineListView::AddItem()** for more information.

The destructor is empty.

Member Functions

- Deselect() see Select()

- DrawItem()

```
virtual void DrawItem(BView *owner,
    BRect itemRect,
    bool drawEverything = false) = 0
```

Hook function that's invoked when the item's *owner* (a BListView object) needs to draw this item. *itemRect* is the area within the owner to which the item's drawing is clipped. If *drawEverything* is **true**, this function should touch every pixel in *itemRect*; if it's **false**, the function can assume that the background color for the item is already painted. **DrawItem()** should be implemented to visually reflect the state of the item, highlighting it if it's selected, dimming it if it's disabled, and so on.

The *owner* view provides the drawing environment for this item. All drawing instructions should be invoked on *owner*. See "BListItem: Creating a Custom List Item" for an example **DrawItem()** function.

- Height() see SetHeight()
- IsEnabled() see SetEnabled()
- IsExpanded() see SetExpanded()
- IsSelected() see Select()

- OutlineLevel()

```
uint32 OutlineLevel(void) const
```

Returns the outline level of the item; this is only meaningful if the item is in a BOutlineListView. See BOutlineListView for more information about levels.

- Select() , Deselect() , IsSelected()

```
void Select(void)
void Deselect(void)
bool IsSelected(void) const
```

Select() and **Deselect()** set the item's selected state; **IsSelected()** returns the state. The setting functions don't automatically update the item's display: After calling **Select()** or **Deselect()**, you must tell the owner to redrawn this item (see **BListView::InvalidateItem()**).

Also, **Select()** doesn't deselect the current selection. If this item is part of a single-selection list view, you have to deselect the current selection yourself.

If possible, you should use BListView's versions of these functions (see **BListView::Select()**). They update the view and manage the selection for you.

SetEnabled() , IsEnabled()

void **SetEnabled(**bool *enabled***)**
bool **IsEnabled(**void**)** const

SetEnabled() sets the list item's enabled state; **IsEnabled()** returns the state. A disabled item can't be selected by the user or by **BListView::Select()**, but it *can* be selected through **BListItem::Select()**. If an item is already selected, **SetEnabled(false)** doesn't deselect it.

After calling **SetEnabled()**, you must tell the owner list view to redraw the item (see **BListView::InvalidateItem()**).

Note that BListView doesn't provide smart versions of these functions (as it does for **Select()**).

SetExpanded() , IsExpanded()

void SetExpanded(bool *expanded***)**
bool IsExpanded(void) const

These functions set and return the item's expanded state. This is only meaningful if the item is part of a BOutlineListView. The item is not automatically redrawn; you must tell the BOutlineListView to redraw the item (and its sublist) through **BOutlineListView::Invalidate()**.

If possible, you should use BOutlineListView's **Expand()** and **Collapse()** functions instead of **SetExpanded()**. The BOutlineListView functions redraw the affected part of the list for you.

SetHeight() , SetWidth() , Height() , Width()

void SetHeight(float *height***)**
void SetWidth(float *width***)**
float Height(void) const
float Width(void) const

These functions set and return the width and height of the item. The item's dimensions are adjusted when `Update()` is called.

Update()

```
virtual void Update(BView *owner, const BFont *font)
```

Hook function that's called whenever the item's *owner* changes in a way that could affect the appearance of this item. It's always called when the item is added to a list view.

The default implementation sets the item's width to that of the owner, and sets its height so it will accommodate *font* (the owner's current BFont setting).

Width() see SetHeight()

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_sel"	`B_BOOL_TYPE`	`true` if the item is selectable.
"_disable"	`B_BOOL_TYPE`	`true` if the item is disabled.
"_li_expanded"	`B_BOOL_TYPE`	`true` if the item is expanded.
"_li_outline_level"	`B_INT32_TYPE`	The outline level of the item.

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the item is not selectable, the "_sel" field won't be found in the archive.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BListItem Function Summary

Function Summary

Function	Description
Deselect()	Deselects the item
DrawItem()	Hook to draw the item
Height()	Returns item height
IsEnabled()	Tests whether the item is enabled
IsExpanded()	Tests whether the item is expanded
IsSelected()	Tests whether the item is selected
BListItem()	Constructor
~BListItem()	Archive constructor
OutlineLevel()	Returns outline level of the item
Select()	Selects the item
SetEnabled()	Sets enabled flag of item
SetExpanded()	Expands or collapses item
SetHeight()	Sets item height
SetWidth()	Sets item width
Update()	Hook called when item is modified
Width()	Returns item width

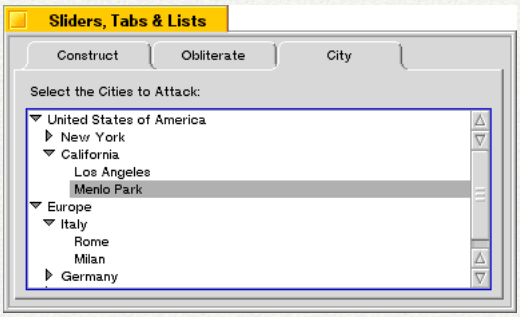
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BOutlineListView

Derived from: public BListView
Declared in: be/interface/OutlineListView.h
Library: libbe.so
Summary: *more...*

A BOutlineListView displays a list of items that can be structured like an outline, with items grouped under other items. The levels of the outline are indicated by successive levels of indentation.



The outline list view shown above was created using the following code:

```
BOutlineListView *outline;
BListItem *region;
BListItem *state;

SetViewColor(216,216,216,0);

r = Bounds();
r.InsetBy(5,5);
r.right -= 16;
r.top += 20;

outline = new BOutlineListView(r, "cities_list",
               B_MULTIPLE_SELECTION_LIST);
```

First, the BOutlineListView is created, with a rectangle computed by insetting from the parent view's bounds rectangle. The **B_MULTIPLE_SELECTION_LIST** flag is specified to indicate that the user should be allowed to choose more than one item in the list.

```
outline->AddItem(region = new BStringItem("United States of America"));
```

This creates a new item in level 0 (the "region" level).

The United States is then divided into states, which comprise level 1 of the outline list. This line of code adds California to the list, placing it "under" the `region` (United States of America):

```
outline->AddUnder(state = new BStringItem("California"), region);
```

a pointer to the new item for California is saved in the variable `state`.

```
outline->AddUnder(new BStringItem("Menlo Park"), state);
outline->AddUnder(new BStringItem("Los Angeles"), state);
```

California is then further divided into cities: Menlo Park and Los Angeles, which reside at level 2 of our outline list. These are inserted under the California item by specifying the pointer to that item (`locality`) when calling **`AddUnder()`**.

This process is repeated for New York state, which has three cities available in our list:

```
outline->AddUnder(locality = new BStringItem("New York"), region);
outline->AddUnder(new BStringItem("Albany"), locality);
outline->AddUnder(new BStringItem("Buffalo"), locality);
outline->AddUnder(new BStringItem("New York City"), locality);
```

Then the Europe region is added (in level 0), and the nations of France, Germany, and Italy are added as localities (level 1). Each of those three localities has cities, which are added into level 2.

```
outline->AddItem(region = new BStringItem("Europe"));
outline->AddUnder(locality = new BStringItem("France"), region);
outline->AddUnder(new BStringItem("Paris"), locality);

outline->AddUnder(locality = new BStringItem("Germany"), region);
outline->AddUnder(new BStringItem("Berlin"), locality);
outline->AddUnder(new BStringItem("Hamburg"), locality);

outline->AddUnder(locality = new BStringItem("Italy"), region);
outline->AddUnder(new BStringItem("Milan"), locality);
outline->AddUnder(new BStringItem("Rome"), locality);
```

Once the list has been completely constructed, a BScrollView is created to contain the outline list view, and is then added to the list. See the BScrollView class for details on how this works:

```
AddChild(new BScrollView("scroll_cities", outline,
        B_FOLLOW_LEFT|B_FOLLOW_TOP, 0, false, true));
```

Finally, a BStringView is created to label the list with a prompt indicating that you should "Select the Cities to Attack:".

```
r = Bounds();
```

```
    r.InsetBy(5,5);
    r.bottom = r.top + 12;
    AddChild(new BStringView(r, "message_string",
          "Select the Cities to Attack:"));
```

Outline Structure

If an item has other items under it—that is, if the immediately following item in the list is at a deeper level of the outline—it is a *superitem*; the items grouped under it are its *subitems*. Superitems are marked by a triangular icon or *latch*, in the usual interface for hypertext lists.

The user can collapse or expand sections of the outline by manipulating the latch. When a section is collapsed, only the superitem for that section is visible (and the latch points to the superitem). All items that follow the superitem are hidden, up to the next item that's not at a deeper outline level. When a section is expanded, subitems are visible (and the latch points downward).

Inherited Functions

The BOutlineListView class inherits most of its functionality from the BListView class. However, inherited functions are concerned only with the expanded sections of the list, not with sections that are hidden because they're collapsed. If an inherited function returns an index or takes an index as an argument, the index counts just the items that are shown on-screen (or could be shown on-screen if they were scrolled into the visible region of the view). `DoForEach()` skips items that can't be displayed. `CountItems()` counts items only in the expanded sections of the list.

However, the functions that the BOutlineListView class itself defines are concerned with all sections of the list, expanded or collapsed. For its functions, an index counts all items in the list, whether visible or not.

The class defines some functions that match those it inherits, but its versions prefix `FullList`... to the function name and don't ignore any items. For example, `FullListCountItems()` counts every item in the list and `FullListDoForEach()` doesn't skip items in collapsed sections.

In some cases, BOutlineListView simply overrides an inherited function without adding the `FullList`... prefix. You should always use the BOutlineListView versions of these functions, not the BListView versions. For example, BOutlineListView's version of `MakeEmpty()` truly empties the list; BListView's version would remove items from the screen, but not from the real list.

Constructor and Destructor

BOutlineListView()

```
BOutlineListView(BRect frame, const char *name, list_view_type type = B_SINGLE_SELECTION_
uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP, uint32 flags =      B_WILL_DRAW |
B_FRAME_EVENTS | B_NAVIGABLE)
BOutlineListView(BMessage *archive)
```

Initializes the BOutlineListView. This constructor matches the BListView constructor in every detail, including default arguments. All argument values are passed to the BListView constructor without change. The BOutlineListView class doesn't do any initialization of its own.

See also: the BListView constructor

~BOutlineListView()

```
virtual ~BOutlineListView()
```

Does nothing; this class relies on the BListView destructor.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BOutlineListView object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, this function returns `NULL` if the specified *archive* doesn't contain data for a BOutlineListView object.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

AddItem() , AddUnder()

```
virtual bool AddItem(BListItem *item)
virtual bool AddItem(BListItem *item, int32 index)
virtual bool AddUnder(BListItem *item, BListItem *superitem)
```

These functions add an *item* to the list. `AddItem()` adds the item at *index*—where the index counts all items assigned to the BOutlineListView—or, if an *index* isn't specified, at the end of the list. The two versions of this function override their BListView counterparts to ensure that the item is correctly entered into the outline. If the item is added to a portion of the list that is collapsed, it won't be visible.

`AddUnder()` adds an *item* immediately after another item in the list and at one outline level deeper. The level of the *item* is modified accordingly. Thus, the item already in the list becomes the superitem for the newly added *item*. If its new superitem is collapsed or is in a collapsed part of the list, the item will not be visible.

Unlike `AddUnder()`, `AddItem()` respects the outline level of the item. By setting the item's level before calling `AddItem()`, you can add it as a subitem to an item at a higher outline level or insert it as a superitem to items at a lower level.

See also: the BListItem class

AddList()

```
virtual bool AddList(BList *newItems)
virtual bool AddList(BList *newItems, int32 index)
```

Adds a group of items to the list just as `AddItem()` adds a single item. The index counts all items assigned to the BOutlineListView. The *newItems* BList must contain pointers to BListItem objects.

See also: `BListView::AddList()`

AddUnder() see AddItem()

Archive()

```
virtual status_t Archive(BMessage *archive, bool deep = true) const
```

Archives the BOutlineListView object much as the **Archive()** function in the BListView class does, but makes sure that all items are archived, including items in collapsed sections of the list, when the *deep* flag is true.

See also: **BListView::Archive()**, **Instantiate()** static function

---

Collapse() , Expand()

void **Collapse(**BListItem ***item*)**
void **Expand(**BListItem ***item*)**

These functions collapse and expand the section of the list controlled by the *item* superitem. If *item* isn't a superitem, it is nevertheless flagged as expanded or collapsed so that it will behave appropriately in case it does become a superitem.

See also: **BListItem::SetExpanded()**

CountItemsUnder()

int32 **CountItemsUnder(**BListItem ***underItem*, bool *oneLevelOnly*) **const**

Counts the items located under the *underItem* superitem and returns that value. If the *oneLevelOnly* argument is **true**, only items directly contained within the specified superitem are considered. If *oneLevelOnly* is **false**, all subitems of the specified item are considered, as are all subitems of those subitems and so forth.

When *oneLevelOnly* is **false**, `CountItemsUnder()` acts just like **FullListCountItems()**, except the first item in the list that's considered is *underItem* instead of the first item in the full list.

See also: **BListItem::CountItems()**

EachItemUnder()

BListItem ***EachItemUnder(**BListItem ***underItem*, bool *oneLevelOnly*, BListItem *(**eachFunc*) (BLi void *), void **data*)

Calls the function *eachFunc* for every item located under the *underItem* superitem. If *oneLevelOnly* is **true**, *eachFunc* is only called for items located directly under the *underItem*; if *oneLevelOnly* is **false**, the *eachFunc* is called once for every subitem of *underItem*, including subitems of subitems, recursively.

The *data* argument is passed through to each call to the *eachFunc* function as the second argument to that function, and may be used for whatever purpose your *eachFunc* requires.

If the *eachFunc* function returns a pointer to a BListItem (rather than **NULL**), processing is stopped immediately, even if there are items still unvisited. The BListItem pointer returned by the *eachFunc* function is returned by `EachItemUnder()`.

When *oneLevelOnly* is **false**, `EachItemUnder()` acts just like **`FullListDoForEach()`**, except the first item in the list that's considered is *underItem* instead of the first item in the full list.

See also: **`BListItem::DoForEach()`**

FullListCountItems() , FullListCurrentSelection() , FullListFirstItem() , FullListLastItem() , FullListIndexOf() , FullListItemAt() , FullListHasItem() , FullListIsEmpty() , FullListDoForEach() , FullListSortItems()

int32 **`FullListCountItems`**`(`void`)` const
int32 **`FullListCurrentSelection`**`(`int32 *index* = 0`)` const
BListItem ***`FullListFirstItem`**`(`void`)` const
BListItem ***`FullListLastItem`**`(`void`)` const
int32 **`FullListIndexOf`**`(`BPoint *point*`)` const int32 **`FullListIndexOf`**`(`BListItem ****item*`)` const
BListItem ***`FullListItemAt`**`(`int32 *index*`)` const
bool **`FullListHasItem`**`(`BListItem ****item*`)` const
bool **`FullListIsEmpty`**`(`void`)` const
void **`FullListDoForEach`**`(`bool (**func*)(BListItem *)`)` void **`FullListDoForEach`**`(`bool (**func*)(BListItem *, void *), void ***`)`
void **`FullListSortItems`**(int (**compareFunc*) (const BListItem *, const BListItem *))

These functions parallel a similar set of functions defined in the BListView class. The BListView functions have identical names, but without the `FullList`... prefix. When applied to a BOutlineListView object, the inherited functions consider only items in sections of the outline that can be displayed on-screen—that is, they skip over items in collapsed portions of the list.

These BOutlineListView functions, on the other hand, consider all items in the list. For example, **`IndexOf()`** and **`FullListIndexOf()`** both return an index to a given *item*. However, for **`IndexOf()`** the index is to the position of the item in the list that can be currently displayed, but for **`FullListIndexOf()`** it's to the item's position in the full list, including collapsed sections.

IsExpanded()

bool **`IsExpanded`**`(`int32 *index*`)`

Returns `true` if the item at *index* is marked as controlling an expanded section of the list, and `false` if it's marked as controlling a collapsed section or if there's no item at that index. If a superitem is expanded, the BOutlineListView can display its subitems; if not, the subitems are hidden.

The *index* passed to this function is to the full list of items assigned to the BOutlineListView.

See also: `BListItem::IsExpanded()`

ItemUnderAt()

BListItem ***ItemUnderAt(**BListItem ***underItem**, bool *oneLevelOnly*, int32 *index***) const**

Returns a pointer to the *index*th item under the specified *underItem*. If *oneLevelOnly* is **true**, only items located directly under the *underItem* are considered. If *oneLevelOnly* is **false**, subitems are scanned recursively to locate the appropriate *index*.

When *oneLevelOnly* is **false**, `ItemUnderAt()` acts just like `FullListItemAt()`, except the first item in the list that's considered is *underItem* instead of the first item in the full list.

KeyDown()

virtual void **KeyDown(**const char ***bytes**, int32 *numBytes***)**

Augments the inherited version of `KeyDown()` to allow users to navigate the outline hierarchy using the arrow keys and to expand or collapse sections of the outline using Control–arrow key combinations.

See also: `BListView::KeyDown()`

MakeEmpty()

virtual void **MakeEmpty(**void**)**

Overrides the BListView version of `MakeEmpty()` to remove all items from the list. The BListView version of this function won't work as advertised on a BOutlineListView.

See also: `BListView::MakeEmpty()`

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Augments the inherited version of **MouseDown()** to permit users to expand and collapse sections of the outline by clicking on an item's latch.

See also: **BView::MouseDown()**

RemoveItem() , RemoveItems()

virtual bool **RemoveItem(**BListItem ***item*)**
virtual BListItem ***RemoveItem(**int32 *index*)**

virtual bool **RemoveItems(**int32 *index*, int32 *count*)**

These functions work like their BListView counterparts, except that:

- They can remove items from any part of the list, including collapsed sections. The *index* counts all items assigned to the BOutlineListView; the specified *item* can be hidden.
- If the item being removed is a superitem, they also remove all of its subitems.

The BListView versions of these functions will not produce reliable results when applied to a BOutlineListView, even if the item being removed is in an expanded section of the list and is not a superitem.

See also: **BListView::RemoveItem()**

SortItemsUnder()

void **SortItemsUnder(**BListItem ***underItem*, bool *oneLevelOnly*, int (**compareFunc*) (const BListIt
const BListItem *)**)**

Sorts the items located under the specified *underItem*. If *oneLevelOnly* is **true**, only items located directly under the *underItem* are considered. If *oneLevelOnly* is **false**, subitems are scanned recursively and sorted.

The hierarchy of the list is ignored when sorting with *oneLevelOnly* set to **false**; items can (and will) be moved from one level to another. If you want to sort without moving items from one level to another, you should call `SortItemsUnder()` once for each superitem with *oneLevelOnly* set to **true**.

When *oneLevelOnly* is **false**, `SortItemsUnder()` acts just like **FullListSortItems()**, except the first item in the list that's considered is *underItem* instead of the first item in the full list.

Superitem()

BListItem ***Superitem(**const BListItem ***item*)**

Returns the superitem for the *item* passed as an argument—that is, the item under which the argument *item* is grouped—or `NULL` if the *item* is at the outermost level of the outline (level 0) or isn't in the list.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BOutlineListView Function Summary

Function Summary

Function	Description
AddItem()	Adds BListItem, preserving the item depth
AddList()	Adds a group of items to the list
AddUnder()	Adds BListItem after a given item and one level deeper
Archive()	Archives a BOutlineListView into a BMessage
Collapse()	Collapses section of the list
CountItemsUnder()	Count the number of items located under a given item
EachItemUnder()	Call a function on each item located under a given item
Expand()	Expands section of the list
FullListCountItems()	CountItems() applied to entire list
FullListCurrentSelection()	CurrentSelection() applied to entire list
FullListDoForEach()	DoForEach() applied to entire list
FullListFirstItem()	FirstItem() applied to entire list
FullListHasItem()	HasItem() applied to entire list
FullListIndexOf()	IndexOf() applied to entire list
FullListIsEmpty()	IsEmpty() applied to entire list
FullListItemAt()	ItemAt() applied to entire list
FullListLastItem()	LastItem() applied to entire list

FullListSortItems()	SortItems() applied to entire list
Instantiate()	Creates a BOutlineListView from a BMessage
IsExpanded()	Returns true if given item is expanded
ItemUnderAt()	Returns the nth item under a specified item
KeyDown()	Provided for implementation
MakeEmpty()	Removes all items from the list
MouseDown()	Provided for implementation
BOutlineListView()	Constructor
~BOutlineListView()	Destructor
RemoveItem()	Remove a single item and its subitems from the list
RemoveItems()	Remove multiple items and their subitems from the list
SortItemsUnder()	Sorts the items located under a given item
Superitem()	Returns the superitem for a given BListItem

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStringItem

Derived from: public BListItem
Declared in: be/interface/ListItem.h
Library: libbe.so
Summary: *more...*

A BStringItem is a container for text that you want to display as an item in a BListView. To use a BStringItem, you construct it, set its text (either in the constructor or through **SetText()**) and then add it to a BListView object.

Constructor and Destructor

BStringItem()

```
BStringItem(const char *text, uint32 level = 0, bool expanded = true)
BStringItem(BMessage *archive)
```

Initializes the BStringItem by making a copy of the *text* string passed as an argument. This is the string the item will display. The *level* and *expanded* arguments are passed unchanged to the BListItem constructor; see that function for an explanation.

~BStringItem()

```
virtual ~BStringItem()
```

Frees the text the item displays.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BStringItem object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain archived data for a BStringItem, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()** and stores the BStringItem in the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

DrawItem()

virtual void **DrawItem(**BView **owner*, BRect *frame*, bool *complete* = false**)**

Draws the text string, dimming it if the item is disabled and highlighting it if the item is selected.

See also: **BListItem::DrawItem()**

SetText() , Text()

virtual void **SetText(**const char **text***)**

const char ***Text(**void**)** const

These functions set and return the text that the BStringItem draws. **SetText()** copies the string it's passed. **Text()** returns a pointer to the string owned by the BStringItem.

Update()

virtual void **Update(**BView **owner*, const BFont **font***)**

Overrides the BListItem version of **Update()** to recalculate the width and height of the BStringItem and the placement of the text. The width of the item is based on the width of the *owner* BView. The height and text placement are based on the owner's *font*. The item must be tall enough to display the string in the current font.

See also: **BListItem::Update()**

Archived Fields

The `Archive()` function adds the following field to its BMessage argument:

Field	Type code	Meaning
"_label"	`B_STRING_TYPE`	String text (present only if it is not `NULL`).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStringItem Function Summary

Function Summary

Function	Description
Archive()	Archives the BStringItem into a BMessage
DrawItem()	Draws the text string
Instantiate()	Creates a BStringItem from a BMessage
SetText()	Sets the text string
BStringItem()	Constructor
~BStringItem()	Destructor
Text()	Returns the text string
Update()	Recalculates the object's dimensions and placement

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStringView

Derived from: public BView
Declared in: be/interface/StringView.h
Library: libbe.so
Summary: *more...*

A BStringView object draws a string that the user can neither select nor edit. BStringView object are typically used as labels for your app's UI. If you need to present text that can be selected and/or edited, wse a BTextView object.

Constructor and Destructor

BStringView()

```
BStringView(BRect frame, const char *name, const char *text,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW)
BStringView(BMessage *archive)
```

Initializes the BStringView by assigning it a *text* string and the system plain font (**be_plain_font**). The *frame*, *name*, *resizingMode*, and *flags* arguments are passed unchanged to the BView constructor.

The *frame* rectangle needs to be large enough to display the entire string in the current font. The string is drawn at the bottom of the frame rectangle and, by default, is aligned to the left side. A different horizontal alignment can be set by calling **SetAlignment()**.

~BStringView()

```
virtual ~BStringView()
```

Frees the text string.

Static Functions

- Instantiate() see BArchivable::Instantiate()

Member Functions

Alignment() see SetAlignment()

Archive() , see

virtual status_t **Archive(**BMessage ***archive**, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, then adds the string and its alignment to the BMessage *archive*.

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets the BStringView's low color and its background view color to match the background color of its new parent view.

See also: **BView::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the string along the bottom of the BStringView's frame rectangle in the current high color.

SetAlignment() , Alignment()

void **SetAlignment(**alignment *flag***)**
alignment **Alignment(**void**)** const

These functions align the string within the BStringView's frame rectangle and return the current alignment. The alignment *flag* can be:

B_ALIGN_LEFT	The string is aligned at the left side of the frame rectangle.
B_ALIGN_RIGHT	The string is aligned at the right side of the frame rectangle.
B_ALIGN_CENTER	The string is aligned so that the center of the string falls midway between the left and right sides of the frame rectangle.

The default is **B_ALIGN_LEFT**.

SetText() , Text()

void **SetText(**const char **string***)**
const char ***Text(**void**)** const

These functions set and return the text string that the BStringView draws. **SetText()** frees the previous string and copies *string* to replace it. **Text()** returns the null-terminated string.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_text"	`B_STRING_TYPE`	Text displayed in the view (present only if it is not `NULL`).
"_align"	`B_INT32_TYPE`	Alignment of text in frame.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStringView Function Summary

Function Summary

Function	Description
Alignment()	Returns the string alignment within the frame
Archive()	Archives a BStringView into a BMessage
AttachedToWindow()	Provided for implementation
Draw()	Provided for implementation
SetAlignment()	Sets the string alignment within the frame
SetText()	Sets the text
BStringView()	Constructor
~BStringView()	Destructor
Text()	Returns the text

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

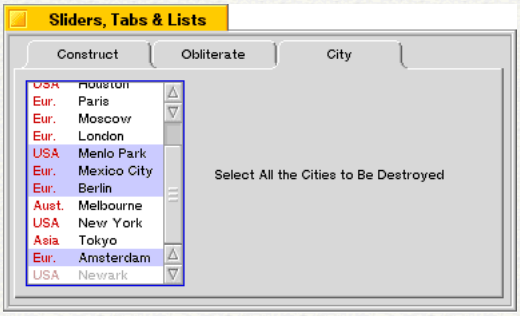
Copyright © 2000 Be, Inc. All rights reserved..

BListItem: Creating a Custom List Item

Derived from: none
Mix-in classes: BArchivable
Declared in: be/interface/ListItem.h
Library: libbe.so
Allocation: Constructor only
more...

Although much of the time all you need to draw in a list are strings (in which case you can use the BStringItem class), from time-to-time you may need to display more than a simple text string—maybe you need to display multiple pieces of information per item, or maybe you just want to jazz up the display with some icons.

For example, let's say you need to let the user select a city from a list, but also need to display the part of the world that each city is in. You could just use BStringItem objects with strings like "Chicago (USA)", but it might look nicer if you could lay out your list items in two colums, maybe with a splash of color:



To change the appearance of a list item, you override the `DrawItem()` function to draw the item's contents however you want it to look.

The following sections define the class that creates these list items.

The CityItem Declaration

The declaration for our "CityItem" class looks like this:

```
#include <String.h>
#include <ListItem.h>

class CityItem : public BListItem
{
   public:
      CityItem(const char *city, int32 region = 0);
      virtual void DrawItem(BView *owner,
            BRect frame,
            bool complete = false);
      enum { USA, ASIA, EUROPE, AUSTRALIA, OTHER };

   private:
      BString kCity;
      int32 kRegion;
};

const char *region_names[] = {
   "USA", "Asia", "Eur.", "Aust.", "Other"
};
```

A CityItem object contains two pieces of data: a city name, and a region code. The region code is used as an index into the array of region names.

The CityItem Definition

The constructor looks like this:

```
CityItem::CityItem(const char *city, int32 region)
         : BListItem()
{
   kCity = city;
   kRegion = region;
}
```

The **DrawItem()** function does the actual work of drawing the item. **DrawItem()** receives three parameters:

- A BView "owner"; this is the view that contains the BListItem. All drawing calls you issue should be made through this BView. For example:

```
owner->DrawString(item_text);
```

- A BRect, which is the rectangle in which the item should be drawn.
- A **bool**, which is **true** if the item needs to be erased and redrawn, or **false** if the item's contents can be safely redrawn without erasing the current contents.

DrawItem() begins by checking to see if the item is selected (by calling **IsSelected()**) or if a *complete* redraw is required. In either of these cases, we want to redraw the background, to either the highlight color, or the owner's view color:

```
void CityItem::DrawItem(BView *owner, BRect frame, bool complete)
{
   if (IsSelected() || complete) {
      rgb_color color;
      if (IsSelected()) {
         color = kHighlight;
      }
      else {
         color = owner->ViewColor();
      }
      owner->SetHighColor(color);
      owner->FillRect(frame);
   }
```

Now we draw the text. First, we move the owner view's pen so it's inset from the bottom left corner of the item's frame. (In a real application, you would want to make the inset adjustments based on the font that's being used; see the BFont class for more information.):

```
   owner->MovePenTo(frame.left+4, frame.bottom-2);
```

If the item is enabled (selectable), we set the owner view's high color to a shade of medium red; if it's disabled, we use a lighter red color (the color definitions aren't shown). Then we use `DrawString()` to draw the region name:.

```
   if (IsEnabled()) {
      owner->SetHighColor(kRedColor);
   }
   else {
      owner->SetHighColor(kDimRedColor);
   }
   owner->DrawString(region_names[kRegion]);
```

Move the pen to the right column and draw the city name:

```
   owner->MovePenTo(frame.left+38, frame.bottom-2);
   if (IsEnabled()) {
      owner->SetHighColor(kBlackColor);
   }
   else {
      owner->SetHighColor(kMedGray);
   }
   owner->DrawString(kCity.String());
}
```

Using a CityItem Object

To use a CityItem object, simply construct a new object and pass it to **BListView::AddItem()**:

```
listView->AddItem(new CityItem("Chicago", CityItem::USA));
```

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BListItem: Creating a Custom List Item Summary

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Scripting Messages

The scripting system defines four generic messages that can operate on the specific properties of an object and two meta-messages that query an object about the messages it can handle. See "Scripting" in **The Application Kit** chapter for a full explanation.

B_COUNT_PROPERTIES

This message requests the number of properties supported by the receiver. It contains no data, but the reply message should contain one field:

Field	Type code	Description
"result"	B_INT32_TYPE	The number of properties supported.

B_GET_SUPPORTED_SUITES

This message requests the names of all message suites that the receiver supports. It doesn't contain any data, but the message that's sent in reply has one field:

Field	Type code	Description
"suites"	B_STRING_TYPE	An array of suite names.

A suite is a named set of messages and specifiers. A BHandler supports the suite if it can respond to the

messages and resolve the specifiers.

B_SET_PROPERTY , B_GET_PROPERTY , B_CREATE_PROPERTY , B_DELETE_PROPERTY

These messages—as their names state—target a particular property under the control of the target handler. They have the following data fields:

Field	Type code	Description
"specifiers"	`B_MESSAGE_TYPE`	An array of one or more BMessages that specify the targeted property. See `AddSpecifier()` in the BMessage class of the Application Kit for details on the contents of a specifier.
"data"	*variable*	For `B_SET_PROPERTY` messages only, the data that should be set. The data type depends on the targeted property.

A class can choose to respond to these messages, in any combination, for any set of self-declared properties.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Scripting

Scripting provides a means for programatically controlling some other application by sending it special scripting commands. These commands are defined by the "scripted" application itself. For example, if you want some other application to be able to tell your application to perform the "FlipOver" operation, you have to publish the format of the "FlipOver" command. The set of operations that you want to expose is called a "suite."

The BeOS defines some number of suites that correspond to particular classes. For example, all BApplication objects respond to the commands defined in the "vnd.Be-application" suite. One of the commands in the suite gives you access to the application's windows. When you've located the window that you want, you can move it, close it, resize it, and so on, according to the commands in the "vnd.Be-window" suite.

Basics

The scripting framework defines the following notions: commands, properties, and specifiers. If you are familiar with AppleScript, these are equivalent to verbs, nouns, and adjectives. Commands act on a specific instance of a property, as determined by the specifiers.

Commands

The command conveys the action of a scripting command and is stored in the *what* field of the scripting BMessage. There are six standard commands (defined in **be/app/Message.h**):

- **`B_COUNT_PROPERTIES`** counts the number of instances of a property.
- **`B_CREATE_PROPERTY`** creates a new instance of a property.
- **`B_DELETE_PROPERTY`** destroys an instance of a property.
- **`B_EXECUTE_PROPERTY`** executes an instance of a property.
- **`B_GET_PROPERTY`** gets the value of an instance of a property.
- **`B_SET_PROPERTY`** sets of the value of an instance of a property. The "data" field contains the new value of the property.

Each of these commands acts on a "property," which is nothing more than a scriptable feature of an object. As a real world example, the windows owned by an application are properties, as is the title of each window. The particular interpretation of the command depends upon the property it is acting on. For example, **`B_DELETE_PROPERTY`**, acting on the "Entry" property of a Tracker window, causes a file to be moved to the trash. However, the same command acting on the "Selection" property of the same window removes files from the list of selected items.

Scriptable objects should limit themselves to this set of commands. If an object uses a nonstandard command, it runs the risk of being unusable by general scripting tools.

Properties and Specifiers

A property represents a scriptable feature of an object. Properties are named; these names are strings unique within a class. For example, a BWindow defines properties such as "Frame," "Title," and "View." The data type of the property and its allowable values are determined by the property. For example, the window's "Frame" accepts **`B_RECT_TYPE`** values while the "Title" is a **`B_STRING_TYPE`**.

Sometimes a property is represented by another object. For example, BWindow's "View" designates a BView, an object which has a set of properties distinct from those of BWindow.

An object may have more than one instance of a given property. For example, the "Window" property of BApplication, has as many instances as there are windows in the application. As a result, there is some ambiguity when you ask for *the* Window of an application. Instead, it's more correct to ask for the first Window, or the Window named "Snyder." In other words, a property is not enough to identify a feature; a specific instance must be picked out as well.

Specifiers are used to target ("specify") particular instances of properties. A specifier is a BMessage containing the following elements:

- The name of the property in the "property" field, stored as a **B_STRING_TYPE**.
- The specifier constant, indicating a method of identifying a specific instance of the property, in the *what* field along with any necessary supporting fields.

There are seven standard specifier constants (defined in **<be/app/Message.h>**):

- **B_DIRECT_SPECIFIER** . The property name is sufficient specification by itself, usually because there's only one instance of the property. If there's more than one value for the property, a **B_DIRECT_SPECIFIER** would specify them all.
- **B_NAME_SPECIFIER** . The specifier message has a "name" field of type **B_STRING_TYPE** with the name of a particular instance of the property.
- **B_ID_SPECIFIER** . The specifier message has an "id" field with a unique identifying value of type **int32** for a particular instance of the property.
- **B_INDEX_SPECIFIER** . The specifier message has an **int32** field named "index" with the index to a particular instance of the property.
- **B_REVERSE_INDEX_SPECIFIER** . The index counts backwards from the end of the list.
- **B_RANGE_SPECIFIER** . In addition to an "index" field, the specifier message has an additional **int32** field named "range", identifying "range" items beginning at "index."
- **B_REVERSE_RANGE_SPECIFIER** . The "index" counts from the end of the list backwards. Depending on the kind of data and the message protocol, the "range" may extend toward the front of the list from the index or toward the end of the list. In other words, the index works in reverse, the range may or may not.

As with messages, the precise meaning of a given specifier depends upon the context. Additionally, there may be user-defined (or perhaps more properly object-defined) specifiers. User-defined specifier constants should be greater than **B_SPECIFIERS_END** to prevent conflicts with Be-defined specifiers.

Specifiers are added to the "specifier" field of a scripting message using **BMessage::AddSpecifier()**. There are several variants of this method, including shortcuts for adding **B_DIRECT_SPECIFIER**, **B_INDEX_SPECIFIER**, **B_RANGE_SPECIFIER**, and **B_NAME_SPECIFIER** specifiers. For all other specifiers, you must manually construct the specifier and add it to the scripting message with **AddSpecifier()**. For example, to add a **B_ID_SPECIFIER**:

```
BMessage specifier(B_ID_SPECIFIER); // create a new specifier
specifier.AddInt32("id", 2827); // add the id number to the specifier
message.AddSpecifier(&specifier); // add the specifier to the message
```

You *must* use **AddSpecifier()** to add specifiers to a BMessage; it performs additional scripting support work that **AddMessage()** doesn't.

The Specifier Stack

In general, an application will not be able to obtain a BMessenger for the target object; instead, it'll have to settle for a BMessenger targeting the BApplication of the program containing the desired object. In these cases, a single specifier may be insufficient to target a scripting message. The true power of specifiers lies in their ability to be chained together in the specifier stack.

An example best illustrates the operation of the specifier stack. The following code snippet creates a message that will target the frame of the second view of the window named "egg" in the target application:

```
message.AddSpecifier("Label");
message.AddSpecifier("MenuBar");
message.AddSpecifier("Window", 1);
```

Repeated calls to **AddSpecifier()** build the specifier stack. The order of the calls is very important; the specifiers are evaluated in the opposite order from which they were added. When this message is received by the target application, it will first peel off the third specifier and direct the message to the second window of the application. The BWindow will then peel off the second specifier and direct the message to the window's key menu bar. The first specifier ("Label") is then processed by the BMenuBar. This process is covered in more detail below under **ResolveSpecifier()**.

Replies

A reply is generated for every scripting request. The reply message contains the following fields:

- The *what* data member defaults to **B_REPLY** unless some other constant is appropriate. For example, if the message was not understood, the object responds with a **B_MESSAGE_NOT_UNDERSTOOD** BMessage.
- The **B_INT32_TYPE** field "error" contains the error code for the operation. This field is always present.
- Responses to a successful **B_GET_PROPERTY** request will additionally contain the value or values of the requested property in the "result" array. The data will be of a type appropriate for the property.

Any scriptable objects that you create should also obey the above protocol. Of course, individual objects are free to define their own protocols for relaying additional information in the reply; in these cases, consult the documentation for the class in question.

Creating and Sending Scripting Messages

The scripting facilities of an application can be invoked in three easy steps:

- Set the command constant for the scripting message.
- Construct the specifier stack for the scripting message.
- Send the scripting message to the target application.

Example

Suppose we want to fetch the frame rectangle of the second view of the window titled "egg" in an application with the signature "application/x-fish". The code:

```
BMessage message, reply;
BRect result;

// set the command constant
message.what = B_GET_PROPERTY;

// construct the specifier stack
message.AddSpecifier("Frame"); // B_DIRECT_SPECIFIER
message.AddSpecifier("View", 1); // B_INDEX_SPECIFIER
message.AddSpecifier("Window", "egg"); // B_NAME_SPECIFIER

// send the message and fetch the result
BMessenger("application/x-fish").SendMessage(&message, &reply);
reply.FindRect("result", &result)
```

Short and sweet.

Suites

There is one missing element in the scripting system, namely the ability to query an object for its scripting abilities. This is useful when the controlling application doesn't know the precise type of the object it is scripting. Having a method of discovering the scripting abilities of an object enables more dynamic uses of scripting.

An object's scripting abilities are organized into one or more scripting "suites," a set of supported messages and associated specifiers. A suite is identified by a MIME-like string with the "suite" supertype. For example, BControl implements the "suite/vnd.Be-control" scripting suite. Nothing prevents two objects from implementing the same suite; two sound editors, for example, could have different implementations of a common scripting suite for filtering audio data.

To ask an object for its supported scripting suites, send it a standard scripting message with a **B_GET_PROPERTY** request for the "Suites" property:

```
message.what = B_GET_PROPERTY;
message.AddSpecifier("Suites");

... add remaining specifiers here ...

messenger.SendMessage(&message, &reply);
```

The target object responds with a **B_REPLY** BMessage with the following fields:

- The error code in "error".
- An array named "suites" containing the names of the suites supported by the object.
- An array named "messages" containing flattened BPropertyInfo objects describing the supported messages and specifiers for the various supported suites.

Less usefully, you can send a **B_GET_SUPPORTED_SUITES** BMessage directly to an object and obtain its supported suites in an identically-formed reply.

Every scriptable object supports the "suite/vnd.Be-handler" suite by dint of its BHandler heritage. This suite is sometimes referred to as the "universal suite." It performs the following functions:

- Implements the "Suites" propery and responds to **B_GET_SUPPORTED_SUITES** messages, as described above.
- Implements the "Messenger" property, allowing the caller to obtain a BMessenger to the object, simplifying further communication with the object.
- Implements the "InternalName" property, returning the name of the BHandler.

- Responds to any other scripting requests with a `B_MESSAGE_NOT_UNDERSTOOD` BMessage. This is a "catch-all" response after all the other objects in the hierarchy have rejected the scripting request.

Making Objects Scriptable

Since scripting messages are passed via BMessengers, objects accepting scripting messages must be derived from BHandler. Typically, adding scripting support entails little more than overriding the following methods:

- **`BHandler::ResolveSpecifier()`** directs the scripting message to the appropriate BHandler.
- **`BHandler::MessageReceived()`** implements the scripting requests.
- **`BHandler::GetSupportedSuites()`** publishes the supported scripting suites.

See the documentation of these functions in the BHandler class for details.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuBar

Derived from: public BMenu
Declared in: be/interface/MenuBar.h
Library: libbe.so
Summary: *more...*

A BMenuBar is a menu that can stand at the root of a menu hierarchy. Rather than appear on-screen when commanded to do so by a user action, a BMenuBar object has a settled location in a window's view hierarchy, just like other views. Typically, the root menu is the menu bar that's drawn across the top of the window. It's from this use that the class gets its name.

However, instances of this class can also be used in other ways. A BMenuBar might simply display a list of items arranged in a column somewhere in a window. Or it might contain just one item, where that item controls a pop-up menu (a BPopUpMenu object). Rather than look like a "menu bar," the BMenuBar object would look something like a button.

The Key Menu Bar

The "real" menu bar at the top of the window usually represents an extensive menu hierarchy; each of its items typically controls a submenu.

The user should be able to operate this menu bar from the keyboard (using the arrow keys and Enter). There are two ways that the user can put the BMenuBar and its hierarchy in focus for keyboard events:

- Clicking an item in the menu bar. If the "click to open" preference is not turned off, this opens the submenu the item controls so that it stays visible on-screen and puts the submenu in focus.
- Pressing the Menu key or Command-Escape. This puts the BMenuBar in focus and selects its first item.

Either method opens the entire menu hierarchy to keyboard navigation.

If a window's view hierarchy includes more than one BMenuBar object, the Menu key (or Command-Escape) must choose one of them to put in focus. By default, it picks the last one that was attached to the window. However, the `SetKeyMenuBar()` function defined in the BWindow class can be called to designate a different BMenuBar object as the "key" menu bar for the window.

A Kind of BMenu

BMenuBar inherits most of its functions from the BMenu class. It reimplements the `AttachedToWindow()`, `Draw()`, and `MouseDown()` functions that set up the object and respond to

messages, but these aren't functions that you'd call from application code; they're called for you.

The only real function (other than the constructor) that the BMenuBar class adds to those it inherits is **`SetBorder()`**, which determines how the list of items is bordered.

Therefore, for most BMenuBar operations—adding submenus, finding items, temporarily disabling the menu bar, and so on—you must call inherited functions and treat the object like the BMenu that it is.

See also: the BMenu class

Constructor and Destructor

BMenuBar()

```
BMenuBar(BRect frame, const char *name,
    uint32 resizingMode = B_FOLLOW_LEFT_RIGHT | B_FOLLOW_TOP,
    menu_layout layout = B_ITEMS_IN_ROW,
    bool resizeToFit = true)
BMenuBar(BMessage *archive)
```

Initializes the BMenuBar by assigning it a *frame* rectangle, a *name*, and a *resizingMode*, just like other BViews. These values are passed up the inheritance hierarchy to the BView constructor. The default resizing mode (**B_FOLLOW_LEFT_RIGHT** plus **B_FOLLOW_TOP**) is designed for a true menu bar (one that's displayed along the top edge of a window). It permits the menu bar to adjust itself to changes in the window's width, while keeping it glued to the top of the window frame.

The *layout* argument determines how items are arranged in the menu bar. By default, they're arranged in a row as befits a true menu bar. If an instance of this class is being used to implement something other than a horizontal menu, items can be laid out in a column (**B_ITEMS_IN_COLUMN**) or in a matrix (**B_ITEMS_IN_MATRIX**).

If the *resizeToFit* flag is turned on, as it is by default, the frame rectangle of the BMenuBar will be automatically resized to fit the items it displays. This is generally a good idea, since it relieves you of the responsibility of testing user preferences to determine what size the menu bar should be. Because the font and font size for menu items are user preferences, items can vary in size from user to user.

When *resizeToFit* is `true`, the *frame* rectangle determines only where the menu bar is located, not how large it will be. The rectangle's `left` and `top` data members are respected, but the `right` and `bottom` sides are adjusted to accommodate the items that are added to the menu bar.

Two kinds of adjustments are made if the *layout* is **B_ITEMS_IN_ROW**, as it typically is for a menu bar:

- The height of the menu bar is adjusted to the height of a single item.
- If the *resizingMode* includes **B_FOLLOW_LEFT_RIGHT**, the width of the menu bar is adjusted to match the width of its parent view. This means that a true menu bar (one that's a child of the window's top view) will always be as wide as the window.

Two similar adjustments are made if the menu bar *layout* is **B_ITEMS_IN_COLUMN**:

- The width of the menu bar is adjusted to the width of the widest item.
- If the *resizingMode* includes **B_FOLLOW_TOP_BOTTOM**, the height of the menu bar is adjusted to match the height of its parent view.

After setting up the key menu bar and adding items to it, you may want to set the minimum width of the window so that certain items won't be hidden when the window is resized smaller.

Change the *resizingMode*, the *layout*, and the *resizeToFit* flag as needed for BMenuBars that are used for a purpose other than to implement a true menu bar.

See also: the BMenu constructor, `BWindow::SetSizeLimits()`

~BMenuBar()

virtual **~BMenuBar()**

Frees all the items and submenus in the entire menu hierarchy, and all memory allocated by the BMenuBar.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BMenuBar object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BMenuBar object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()**, which serves to archive the BMenuBar's current state and, if the *deep* flag is `true`, all its menu items. This function then adds the BMenuBar's border style to the BMessage *archive*.

See also: **BArchivable::Archive()**, **BMenu::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Finishes the initialization of the BMenuBar by setting up the whole hierarchy of menus that it controls, and by making the BWindow to which it has become attached the target handler for all items in the menu hierarchy, except for those items for which a target has already been set.

This function also makes the BMenuBar the key menu bar, the BMenuBar object whose menu hierarchy the user can navigate from the keyboard. If a window contains more than one BMenuBar in its view hierarchy, the last one that's added to the window gets to keep this designation. However, the key menu bar should always be the real menu bar at the top of the window. It can be explicitly set with BWindow's **SetKeyMenuBar()** function.

See also: **BWindow::SetKeyMenuBar()**

Border() see SetBorder()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the menu—whether as a true menu bar, as some other kind of menu list, or as a single item that controls a pop-up menu. This function is called as the result of update messages; you don't need to call it yourself.

See also: **BView::Draw()**

Hide() , Show()

virtual void **Hide**(void)
virtual void **Show**(void)

These functions override their BMenu counterparts to restore the normal behavior for views when they're hidden and unhidden. When an ordinary BMenu is hidden, the window that displays it is also removed from the screen. But it would be a mistake to remove the window that displays a BMenuBar. Hiding a BMenuBar is like hiding a typical view; only the view is hidden, not the window.

See also: `BView::Hide()`

MouseDown()

virtual void **MouseDown**(BPoint *point*)

Initiates mouse tracking and keyboard navigation of the menu hierarchy. This function is called to notify the BMenuBar of a mouse-down event.

See also: `BView::MouseDown()`

SetBorder() , Border()

void **SetBorder**(menu_bar_border *border*)
menu_bar_border **Border**(void) const

`SetBorder()` determines how the menu list is bordered. The *border* argument can be any of three values:

B_BORDER_FRAME	The border is drawn around the entire frame rectangle.
B_BORDER_CONTENTS	The border is drawn around just the list of items.
B_BORDER_EACH_ITEM	A border is drawn around each item.

`Border()` returns the current setting. The default is `B_BORDER_FRAME`.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_border"	`B_INT32_TYPE`	Menu bar border (exists only if not `B_BORDER_FRAME`)

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenu

Derived from: public BView
Declared in: be/interface/Menu.h
Library: libbe.so
Summary: *more...*

A BMenu object displays a pull-down or pop-up list of menu items. A menu can contain simple menu items (BMenuItem objects), or other menus (other BMenus). To add an item to a menu, call `AddItem()`.

Menu Hierarchy

Menus are hierarchically arranged; an item in one menu can control another menu. The controlled menu is a *submenu*; the menu that contains the item that controls it is its *supermenu*. A submenu remains hidden until the user operates the item that controls it; it becomes hidden again when the user is finished with it.

The menu at the root of the hierarchy is displayed in a window as a list—perhaps a list of just one item. Since it, unlike other menus, doesn't have a controlling item, it must remain visible. A root menu is therefore a special kind of menu in that it behaves more like an ordinary view than do other menus, which stay hidden. Root menus should belong to the BMenuBar class, which is derived from BMenu. The typical root menu is a menu bar displayed across the top of a window.

Menu Items

Each item in a menu is a kind of BMenuItem object. An item can be marked (displayed with a check mark to its left), assigned a keyboard shortcut, enabled and disabled, and given a "trigger" character that the user can type to invoke the item when its menu is open on-screen.

Every item has a particular job to do. If an item controls a submenu, its job is to show the submenu on-screen and hide it again. All other items give instructions to the application. When invoked by the user, they deliver a BMessage to a target BHandler. What the item does depends on the content of the BMessage and the BHandler's response to it.

Hook Functions

`ScreenLocation()`
: Can be implemented to have the menu appear on-screen at some location other than the default.

Constructor and Destructor

BMenu()

```
BMenu(const char *name, menu_layout layout = B_ITEMS_IN_COLUMN)
BMenu(const char *name, float width, float height)
BMenu(BMessage *archive)
protected: BMenu(BRect frame, const char *name, uint32 resizingMode, uint32 flags, menu_layout
bool resizeToFit)
```

Initializes the BMenu object. The *name* of the object becomes the initial label of the supermenu item that controls the menu and brings it to the screen. (It's also the name that can be passed to BView's `FindView()` function.)

A new BMenu object doesn't contain any items; you need to call `AddItem()` to set up its contents.

A menu can arrange its items in any of three ways:

B_ITEMS_IN_COLUMN
: The items are stacked vertically in a column, one on top of the other, as in a typical menu.

B_ITEMS_IN_ROW
: The items are laid out horizontally in a row, from end to end, as in a typical menu bar.

B_ITEMS_IN_MATRIX
: The items are arranged in a custom fashion, such as a matrix.

Either `B_ITEMS_IN_ROW` or the default `B_ITEMS_IN_COLUMN` can be passed as the *layout* argument to the public constructor. (A column is the default for ordinary menus; a row is the default for BMenuBars.) This version of the constructor isn't designed for `B_ITEMS_IN_MATRIX` layouts.

A BMenu object can arrange items that are laid out in a column or a row entirely on its own. The menu will be resized to exactly fit the items that are added to it.

However, when items are laid out in a custom matrix, the menu needs more help. First, the constructor must be informed of the exact *width* and *height* of the menu rectangle. The version of the constructor that takes these two parameters is designed just for matrix menus—it sets the layout to `B_ITEMS_IN_MATRIX`. Then, when items are added to the menu, the BMenu object expects to be informed of their precise positions within the specified area. The menu is *not* resized to fit the items that are added. Finally, when items in the matrix change, you must take care of any required adjustments in the layout yourself.

The protected version of the constructor is supplied for derived classes that don't simply devise different sorts of menu items or arrange them in a different way, but invent a different kind of menu. If the *resizeToFit* flag is `true`, it's expected that the *layout* will be

B_ITEMS_IN_COLUMN or B_ITEMS_IN_ROW. The menu will resize itself to fit the items that are added to it. If the layout is B_ITEMS_IN_MATRIX, the *resizeToFit* flag should be `false`.

~BMenu()

virtual **~BMenu()**

Deletes all the items that were added to the menu and frees all memory allocated by the BMenu object. Deleting the items serves also to delete any submenus those items control and, thus, the whole branch of the menu hierarchy.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BMenu object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BMenu object, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

AddItem()

```
bool AddItem(BMenuItem *item)
bool AddItem(BMenuItem *item, int32 index)
bool AddItem(BMenuItem *item, BRect frame)
bool AddItem(BMenu *submenu)
bool AddItem(BMenu *submenu, int32 index)
bool AddItem(BMenu *submenu, BRect frame)
```

Adds an item to the menu list at *index*—or, if no *index* is mentioned, to the end of the list. If items are arranged in a matrix rather than a list, it's necessary to specify the item's *frame* rectangle—the exact position where it should be located in the menu view. Assume a coordinate system for the menu that has the origin, (0.0, 0.0), at the left top corner of the view rectangle. The rectangle will have the width and height that were specified when the menu was constructed.

The versions of this function that take an *index* (even an implicit one) can be used only if the menu arranges items in a column or row (`B_ITEMS_IN_COLUMN` or `B_ITEMS_IN_ROW`); it's an error to use them for items arranged in a matrix. Conversely, the versions of this function that take a *frame* rectangle can be used only if the menu arranges items in a matrix (`B_ITEMS_IN_MATRIX`); it's an error to use them for items arranged in a list.

If a *submenu* is specified rather than an *item*, `AddItem()` constructs a controlling BMenuItem for the submenu and adds the item to the menu.

If it's unable to add the item to the menu—for example, if the *index* is out-of-range or the wrong version of the function has been called—`AddItem()` returns `false`. If successful, it returns `true`.

See also: the BMenu constructor, the BMenuItem class, `RemoveItem()`

AddSeparatorItem()

```
bool AddSeparatorItem(void)
```

Creates an instance of the BSeparatorItem class and adds it to the end of the menu list, returning `true` if successful and `false` if not (a very unlikely possibility). This function is a shorthand for:

```
BSeparatorItem *separator = new BSeparatorItem;
AddItem(separator);
```

A separator serves only to separate other items in the list. It counts as an item and has an indexed position in the list, but it doesn't do anything. It's drawn as a horizontal line across the menu. Therefore, it's appropriately added only to menus where the items are laid out in a column.

See also: `AddItem()`, the BSeparatorItem class

Archive()

```
virtual status_t Archive(BMessage *archive, bool deep = true) const
```

Calls the inherited version of `Archive()`, then archives the BMenu by recording its layout and all current settings in the BMessage *archive*. If the *deep* flag is `true`, all of the menu items are also archived.

See also: `BArchivable::Archive()`, `Instantiate()` static function

AreTriggersEnabled() see SetTriggersEnabled()

AttachedToWindow()

```
virtual void AttachedToWindow(void)
```

Finishes initializing the BMenu object by laying out its items and resizing the BMenu view to fit. This function is called for you each time the BMenu is assigned to a window. For a submenu, that means each time the menu is shown on-screen.

See also: `BView::AttachedToWindow()`

CountItems()

```
int32 CountItems(void) const
```

Returns the total number of items in the menu, including separator items.

Draw()

```
virtual void Draw(BRect updateRect)
```

Draws the menu. This function is called for you whenever the menu is placed on-screen or is updated while on-screen. It's not a function you need to call yourself.

See also: `BView::Draw()`

FindItem()

BMenuItem ***FindItem(**const char **label***)** const
BMenuItem ***FindItem(**uint32 ***command***)** const

Returns the item with the specified *label*—or the one that sends a message with the specified *command*. If there's more than one item in the menu hierarchy with that particular *label* or associated with that particular *command*, this function returns the first one it finds. It recursively searches the menu by working down the list of items in order. If an item controls a submenu, it searches the submenu before returning to check any remaining items in the menu.

If none of the items in the menu hierarchy meet the stated criterion, `FindItem()` returns `NULL`.

FindMarked()

BMenuItem ***FindMarked(**void**)**

Returns the first marked item in the menu list (the one with the lowest index), or `NULL` if no item is marked.

See also: `SetRadioMode()`, `BMenuItem::SetMarked()`

GetMaxContentWidth() see SetMaxContentWidth()

Hide() , Show()

protected: void **Hide(**void**)**

protected: void **Show(**bool ***selectFirst***)**
protected: virtual void **Show(**void**)**

These functions hide the menu (remove the BMenu view from the window it's in and remove the window from the screen) and show it (attach the BMenu to a window and place the window on-screen). If the *selectFirst* flag passed to `Show()` is `true`, the first item in the menu will be selected when it's shown. If *selectFirst* is `false`, the menu is shown without a selected item.

The version of `Show()` that doesn't take an argument simply calls the version that does and passes it a *selectFirst* value of `false`.

These functions are not ones that you'd ordinarily call, even when implementing a derived class. You'd need them only if you're implementing a nonstandard menu of some kind and want to control when the menu appears on-screen.

See also: `BView::Show()`, `Track()`

IndexOf()

```
int32 IndexOf(BMenuItem *item) const
int32 IndexOf(BMenu *submenu) const
```

Returns the index of the specified menu *item*—or the item that controls the specified *submenu*. Indices record the position of the item in the menu list. They begin at 0 for the item at the top of a column or at the left of a row and include separator items.

If the menu doesn't contain the specified *item*, or the item that controls *submenu*, the return value will be `B_ERROR`.

See also: `AddItem()`

InvalidateLayout()

```
void InvalidateLayout(void)
```

Forces the BMenu to recalculate the layout of all menu items and, consequently, its own size. It can do this only if the items are arranged in a row or a column. If the items are arranged in a matrix, it's up to you to keep their layout up-to-date.

All BMenu and BMenuItem functions that change an item in a way that might affect the overall menu automatically invalidate the menu's layout so it will be recalculated. For example, changing the label of an item might cause the menu to become wider (if it needs more room to accommodate the longer label) or narrower (if it no longer needs as much room as before).

Therefore, you don't need to call `InvalidateLayout()` after using a kit function to change a menu or menu item; it's called for you. You'd call it only when making some other change to a menu.

See also: the BMenu constructor

IsEnabled() see SetEnabled()

IsLabelFromMarked() see SetLabelFromMarked()

IsRadioMode() see SetRadioMode()

ItemAt() , SubmenuAt()

```
BMenuItem *ItemAt(int32 index) const
BMenu *SubmenuAt(int32 index) const
```

These functions return the item at *index*—or the submenu controlled by the item at *index*. If there's no item at the index, they return `NULL`. **`SubmenuAt()`** is a shorthand for:

```
ItemAt(index)->Submenu()
```

It returns `NULL` if the item at *index* doesn't control a submenu.

See also: **`AddItem()`**

KeyDown()

virtual void **KeyDown(**const char ***bytes*, int32 *numBytes***)**

Handles keyboard navigation through the menu. This function is called to respond to messages reporting key-down events. It should not be called from application code.

See also: **`BView::KeyDown()`**

Layout()

protected: **`menu_layout`** **Layout(**void**)** const

Returns **`B_ITEMS_IN_COLUMN`** if the items in the menu are stacked in a column from top to bottom, **`B_ITEMS_IN_ROW`** if they're stretched out in a row from left to right, or **`B_ITEMS_IN_MATRIX`** if they're arranged in some custom fashion. By default BMenu items are arranged in a column and BMenuBar items in a row.

The layout is established by the constructor.

See also: the BMenu and BMenuBar constructors

MaxContentWidth() see SetMaxContentWidth()

RemoveItem()

BMenuItem ***RemoveItem(**int32 *index***)**
bool **RemoveItem(**BMenuItem **item***)**
bool **RemoveItem(**BMenu **submenu***)**

Removes the item at *index*, or the specified *item*, or the item that controls the specified *submenu*. Removing the item doesn't free it.

- If passed an *index*, this function returns a pointer to the item so you can free it. It returns a `NULL` pointer if the item couldn't be removed (for example, if the *index* is out-of-range).

- If passed an *item*, it returns `true` if the item was in the list and could be removed, and `false` if not.
- If passed a *submenu*, it returns `true` if the submenu is controlled by an item in the menu and that item could be removed, and `false` otherwise.

When an item is removed from a menu, it loses its target; the cached value is set to `NULL`. If the item controls a submenu, it remains attached to the submenu even after being removed.

See also: `AddItem()`

ResizeToPreferred() see GetPreferredSize()

ScreenLocation()

protected: virtual BPoint **ScreenLocation**(void)

Returns the point where the left top corner of the menu should appear when the menu is shown on-screen. The point is specified in the screen coordinate system.

This function is called each time a hidden menu (a submenu of another menu) is brought to the screen. It can be overridden in a derived class to change where the menu appears. For example, the BPopUpMenu class overrides it so that a pop-up menu pops up over the controlling item.

See also: the BPopUpMenu class

SetEnabled() , IsEnabled()

virtual void **SetEnabled**(bool *enabled*)
bool **IsEnabled**(void) const

`SetEnabled()` enables the BMenu if the *enabled* flag is `true`, and disables it if *enabled* is `false`. If the menu is a submenu, this enables or disables its controlling item, just as if `SetEnabled()` were called for that item. The controlling item is updated so that it displays its new state, if it happens to be visible on-screen.

Disabling a menu disables its entire branch of the menu hierarchy. All items in the menu, including those that control other menus, are disabled.

`IsEnabled()` returns `true` if the BMenu, and every BMenu above it in the menu hierarchy, is enabled. It returns `false` if the BMenu, or any BMenu above it in the menu hierarchy, is disabled.

See also: `BMenuItem::SetEnabled()`

SetItemMargins() , GetItemMargins()

protected: void **SetItemMargins(**float *left*, float *top*, float *right*, float *bottom***)**
protected: void **GetItemMargins(**float **left*, float **top*, float **right*, float **bottom***)**

These functions set and get the margins around each item in the BMenu. For the purposes of this function, you should assume that all items are enclosed in a rectangle of the same size, one big enough for the largest item. Keyboard shortcuts are displayed in the right margin and check marks in the left.

See also: `SetMaxContentWidth()`

SetLabelFromMarked() , IsLabelFromMarked()

protected: void **SetLabelFromMarked(**bool *flag***)**
protected: bool **IsLabelFromMarked(**void**)**

`SetLabelFromMarked()` determines whether the label of the item that controls the menu (the label of the superitem) should be taken from the currently marked item within the menu. If *flag* is `true`, the menu is placed in radio mode and the superitem's label is reset each time the user selects a different item. If *flag* is `false`, the setting for radio mode doesn't change and the label of the superitem isn't automatically reset.

`IsLabelFromMarked()` returns whether the superitem's label is taken from the marked item (but not necessarily whether the BMenu is in radio mode).

See also: `SetRadioMode()`

SetMaxContentWidth() , MaxContentWidth()

virtual void **SetMaxContentWidth(**float *width***)**
float **MaxContentWidth(**void**)** const

These functions set and return the maximum width of an item's content area. The content area is where the item label is drawn; it excludes the margin on the left where a check mark might be placed and the margin on the right where a shortcut character or a submenu symbol might appear. The content area is the same size for all items in the menu.

Normally, a menu will be wide enough to accommodate its longest item. However, items wider than the maximum set by `SetMaxContentWidth()` are truncated to fit.

See also: `SetItemMargins()`, `BMenuItem::TruncateLabel()`

SetRadioMode() , IsRadioMode()

virtual void **SetRadioMode(**bool *flag***)**

bool **IsRadioMode(**void**)**

`SetRadioMode()` puts the BMenu in radio mode if *flag* is `true` and takes it out of radio mode if *flag* is `false`. In radio mode, only one item in the menu can be marked at a time. If the user selects an item, a check mark is placed in front of it automatically (you don't need to call BMenuItem's `SetMarked()` function; it's called for you). If another item was marked at the time, its mark is removed. Selecting a currently marked item retains the mark.

`IsRadioMode()` returns whether the BMenu is currently in radio mode. The default radio mode is `false` for ordinary BMenus, but `true` for BPopUpMenus.

`SetRadioMode()` doesn't change any of the items in the menu. If you want an initial item to be marked when the menu is put into radio mode, you must mark it yourself.

When `SetRadioMode()` turns radio mode off, it calls `SetLabelFromMarked()` and passes it an argument of `false`—turning off the feature that changes the label of the menu's superitem each time the marked item changes. Similarly, when `SetLabelFromMarked()` turns on this feature, it calls `SetRadioMode()` and passes it an argument of `true`—turning radio mode on.

See also: `BMenuItem::SetMarked()`, `SetLabelFromMarked()`

SetTargetForItems()

virtual status_t **SetTargetForItems(**BHandler **handler***)**
virtual status_t **SetTargetForItems(**BMessenger *messenger***)**

Assigns *handler* or *messenger* as the target for all the items in the menu. The proposed target is subject to the restrictions imposed by the `SetTarget()` function that BMenuItem inherits from BInvoker in the Application Kit. See that function for further information.

If it's unable to set the target of any item, `SetTargetForItems()` aborts and returns the error it encountered. If successful in setting the target of all items, it returns `B_OK`.

This function doesn't work recursively (it doesn't descend into submenus), and it only acts on items that are currently in the BMenu (it doesn't affect items that are added later).

SetTriggersEnabled() , AreTriggersEnabled()

virtual void **SetTriggersEnabled(**bool *flag***)**

bool **AreTriggersEnabled(**void**)** const

`SetTriggersEnabled()` enables the triggers for all items in the menu if *flag* is `true` and disables them if *flag* is `false`. `AreTriggersEnabled()` returns whether the triggers are currently enabled or disabled. They're enabled by default.

Triggers are displayed to the user only if they're enabled, and only when keyboard actions can operate the menu.

Triggers are appropriate for some menus, but not for others. `SetTriggersEnabled()` is typically called to initialize the BMenu when it's constructed, not to enable and disable triggers as the application is running. If triggers are ever enabled for a menu, they should always be enabled; if they're ever disabled, they should always be disabled.

See also: `BMenuItem::SetTrigger()`

Show() see Hide()

SubmenuAt() see ItemAt()

Superitem() , Supermenu()

BMenuItem ***Superitem**(void) const
BMenu ***Supermenu**(void) const

These functions return the supermenu item that controls the BMenu and the supermenu where that item is located. The supermenu could be a BMenuBar object. If the BMenu hasn't been made the submenu of another menu, both functions return `NULL`.

See also: `AddItem()`

Track()

protected: BMenuItem ***Track**(bool *openAnyway* = false, BRect **clickToOpenRect* = NULL)

Initiates tracking of the cursor within the menu. This function passes tracking control to submenus (and submenus of submenus) depending on where the user moves the mouse. If the user ends tracking by invoking an item, `Track()` returns the item. If the user didn't invoke any item, it returns `NULL`. The item doesn't have to be located in the BMenu; it could, for example, belong to a submenu of the BMenu.

If the *openAnyway* flag is `true`, `Track()` opens the menu and leaves it open even though a mouse button isn't held down. This enables menu navigation from the keyboard. If a *clickToOpenRect* is specified and the user has set the click-to-open preference, `Track()` will leave the menu open if the user releases the mouse button while the cursor is inside the rectangle. The rectangle should be stated in the screen coordinate system.

`Track()` is called by the BMenu to initiate tracking in the menu hierarchy. You would need to call it yourself only if you're implementing a different kind of menu that starts to track the cursor under nonstandard circumstances.

Scripting Support

The BMenu class implements the suite called "suite/vnd.Be-menu" consisting of the following messages:

The Enabled Property

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Retur ns `true` if menu or menu item is enabled; `false` otherwise.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Enables or disables menu or menu item.

Whether the menu or menu item is enabled or disabled The "Enabled" property reflects whether the menu or menu item is enabled or disabled. ### The Label Property	Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the string label of the menu or menu item.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the string label of the menu or menu item.	

The menu or menu item's text label The "Label" property refers to the text label of a menu or menu item. ### The Mark Property	Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns `true` if the menu item or the menu's superitem is marked; `false` otherwise.	
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Marks or unmarks the menu item or the menu's superitem.	

Whether the menu item or the menu's superitem is	Message	Specifiers	Meaning

marked **The "Mark" property refers to whether or not a given menu item or a given menu's superitem is marked.** **The Menu Property**			
B_CREATE_PROPERTY	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Adds a new menu item at the specified index with the text label found in "data" and the int32 command found in "what" (used as the what field in the BMessage sent by the item).	
B_DELETE_PROPERTY	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Removes the selected menu or menus.	
any other	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Directs scripting message to the specified menu, first popping the current specifier off the stack.	

BMenus contained in the menu **The "Menu" property refers to individual BMenus in the menu.** **The MenuItem Property**	**Message**	**Specifiers**	**Meaning**
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Counts the number of menu items in the specified menu.	
B_CREATE_PROPERTY	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Adds a new menu item at the specified index with the text label found in "data" and the int32 command found in "what" (used as the what field in the BMessage sent by the item).	
B_DELETE_PROPERTY	B_NAME_SPECIFIER,	Removes the specified menu	

	B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	item from its parent menu.
B_EXECUTE_PROPERTY	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Invokes the specified menu item.
any other	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	Directs scripting message to the specified menu, first popping the current specifier off the stack.

BMenuItems contained in the menu **The "MenuItem" property refers to individual BMenuItems in the menu.**	**Field**	**Type code**	**Meaning**

Archived Fields **The Archive() function adds the following fields to its BMessage argument:**			
"_layout"	B_INT32_TYPE	Menu layout (Exists only if layout not B_ITEMS_IN_ROW).	
"_rsize_to_fit"	B_BOOL_TYPE	true if menu resizes to fit items.	
"_disable"	B_BOOL_TYPE	true if menu is disabled.	
"_radio"	B_BOOL_TYPE	true if menu is in radio mode.	
"_trig_disabled"	B_BOOL_TYPE	true if menu triggers are disabled.	
"_dyn_label"	B_BOOL_TYPE	true if menu label mirrors the currently selected item.	
"_maxwidth"	B_FLOAT_TYPE	Maximum content width of the menu.	
"_items" (array)	B_MESSAGE_TYPE	Menu items (only in deep copy).	
"_i_frames" (array)	B_MESSAGE_TYPE	Location of items (only in deep copy of layout B_ITEMS_IN_MATRIX)	

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the menu is disabled, the "_disable" field won't be found in the archive.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenu Function Summary

Function Summary

Function	Description
AddItem()	Adds BMenu or BMenuItem to the menu
AddSeparatorItem()	Adds separator item to the menu
Archive()	Archives a BMenu into a BMessage
AreTriggersEnabled()	Tests if triggers are enabled
AttachedToWindow()	Provided for implementation
CountItems()	Returns number of items in the menu
Draw()	Provided for implementation
FindItem()	Returns a specified BMenuItem
FindMarked()	Returns first marked item in the menu
GetItemMargins()	protected: Returns margins around each menu item
Hide()	protected: Hides the menu
IndexOf()	Returns index of specified menu or submenu
Instantiate()	Creates a BMenu from a BMessage
InvalidateLayout()	Forces BMenu to recalculate the menu layout
IsEnabled()	Tests whether the menu is enabled
IsLabelFromMarked()	protected: Returns label from marked flag

IsRadioMode()	Tests whether the menu is in radio mode
ItemAt()	Returns menu at a specified index
KeyDown()	Provided for implementation
Layout()	protected: Returns menu layout
MaxContentWidth()	Returns maximum width of an item's content area
BMenu()	Constructor
~BMenu()	Destructor
RemoveItem()	Removes specified item or submenu
ScreenLocation()	protected: Hook to position menu
SetEnabled()	Enables or disables menu
SetItemMargins()	protected: Sets margins around each menu item
SetLabelFromMarked()	protected: Sets label from marked flag
SetMaxContentWidth()	Sets maximum width of an item's content area
SetRadioMode()	Enables or disables radio mode for the menu
SetTargetForItems()	Sets target BHandler or BMessenger for all menu items
SetTriggersEnabled()	Enables or disables triggers for all menu items
Show()	protected: Displays the menu
SubmenuAt()	Returns submenu controlled by item at specified index

Superitem()	Returns BMenuItem controlling the menu
Supermenu()	Returns BMenu controlling the menu
Track()	protected: Initiates tracking of cursor within menu

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuItem

Derived from: (none)
Mix-in classes: BInvoker, BArchivable
Declared in: be/interface/MenuItem.h
Library: libbe.so
Summary: *more...*

A BMenuItem object displays one item within a menu and contains the state associated with that item. By default, menu items display text; a derived class can reimplement the `Draw()` and `DrawContent()` hook functions to draw something else.

Each BMenuItem object can have its own invocation message and target. A menu item needn't send any message—it can be used simply for its visual presence (see BSeparatorItem for an example).

Menu items can't be used outside of a menu; to add a BMenuItem to a BMenu, call `BMenu::AddItem()`.

Kinds of Items

Some menu items set up the menu hierarchy by giving users access to submenus. A submenu remains hidden until the user operates the item that controls it.

Other items accomplish specific actions. When the user invokes the item, it sends a message to a target BLooper and BHandler, usually the window where the menu at the root of the hierarchy (a BMenuBar object) is displayed. The action that the item initiates, or the state that it sets, depends entirely on the message and the target's response to it.

Shortcuts and Triggers

Any menu item (except for those that control submenus) can be associated with a keyboard shortcut, a character the user can type in combination with a **Command** key (and possibly other modifiers) to invoke the item. The shortcut character is displayed in the item to the right of the label.

A shortcut works even when the item it invokes isn't visible on-screen. It, therefore, has to be unique within the window (within the entire menu hierarchy).

Every menu item is also associated with a *trigger*, a character that the user can type (without the **Command** key) to invoke the item. The trigger works only while the menu is both open on-screen and can be operated using the keyboard. It therefore must be unique only within a particular branch of the menu hierarchy (within the menu).

The trigger is one of the characters that's displayed within the item—either the keyboard shortcut or a character in the label. When it's possible for the trigger to invoke the item, the

character is underlined. Like shortcuts, triggers are case-insensitive.

For an item to have a keyboard shortcut, the application must explicitly assign one. However, by default, the Interface Kit chooses and assigns triggers for all items. The default choice can be altered by the **SetTrigger()** function.

Marked Items

An item can also be marked (with a check mark drawn to the left of the label) in order to indicate that the state it sets is currently in effect. Items are marked by the **SetMarked()** function. A menu can be set up so that items are automatically marked when they're selected and exactly one item is marked at all times. (See **SetRadioMode()** in the BMenu class.)

Disabled Items

Items can also be enabled or disabled (by the **SetEnabled()** function). A disabled item is drawn in muted tones to indicate that it doesn't work. It can't be selected or invoked. If the item controls a specific action, it won't post the message that initiates the action. If it controls a submenu, it will still bring the submenu to the screen, but all the items in submenu will be disabled. If an item in the submenu brings its own submenu to the screen, items in that submenu will also be disabled. Disabling the superitem for a submenu in effect disables a whole branch of the menu hierarchy.

Hook Functions

`Draw()`

<B><TT>DrawContent()TT>B>

GetContentSize()

<B><TT>Highlight()

TruncateLabel()

Constructor and Destructor

BMenuItem()

```
BMenuItem(const char *label, BMessage *message,
    char shortcut = 0, uint32 modifiers = 0)
BMenuItem(BMenu *submenu, BMessage *message = NULL)
BMenuItem(BMessage *archive)
```

Creates a new BMenuItem and sets its *label*, its invocation *message*, both of which can be **NULL**. When the user invokes the menu item, *message* is copied and the following fields

Whenever the user invokes the item, the model message is copied and the copy is posted and marked for delivery to the target handler. Three pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	B_INT64_TYPE	The time the item was invoked, as measured by the number of microseconds since 12:00:00 AM January 1, 1970.
"source"	B_POINTER_TYPE	A pointer to the BMenuItem object.
"index"	B_INT32_TYPE	The index of the item, its ordinal position in the menu. Indices begin at 0.

By default, the target of the message is the window that the contains the menu that the item is part of. Use SetTarget() to set a different target.

The constructor can also set a keyboard *shortcut* for the item. The character that's passed as the *shortcut* parameter will be displayed to the right of the item's label. Upper case characters are mapped to lower case characters; if you want the user to have to type an upper case character, you must include B_SHIFT_KEY in the modifers mask.

The *modifiers* mask determines which modifier keys the user must hold down for the shortcut to work. The mask can be formed by combining any of the modifiers constants, especially these:

B_SHIFT_KEY
B_CONTROL_KEY
B_OPTION_KEY

B_COMMAND_KEY is required for all keyboard shortcuts; it doesn't have to be explicitly included in the mask.

If the BMenuItem is constructed to control a *submenu*, it can't take a shortcut and it typically doesn't post messages—its role is to bring up the submenu. However, it can be assigned a

model *message* if the application must take some collateral action when the submenu is opened. The item's initial label will be taken from the name of the submenu. It can be changed after construction by calling **SetLabel()**.

~BMenuItem()

virtual **~BMenuItem()**

Frees the item's label and its model BMessage object. If the item controls a submenu, that menu and all its items are also freed. Deleting a BMenuItem destroys the entire menu hierarchy under that item.

Member Functions

ContentLocation()

protected: BPoint **ContentLocation**(void) const

Returns the left top corner of the content area of the item, in the coordinate system of the BMenu to which it belongs. The content area of an item is the area where it displays its label (or whatever graphic substitutes for the label). It doesn't include the part of the item where a check mark or a keyboard shortcut could be displayed, nor the border and background around the content area.

You would need to call this function only if you're implementing a **DrawContent()** function to draw the contents of the menu item. The content rectangle can be calculated from the point returned by this function and the size specified by **GetContentSize()**.

If the item isn't part of a menu, the return value is indeterminate.

Draw() , DrawContent()

protected: virtual void **Draw**(void)

protected: virtual void **DrawContent**(void)

These functions draw the menu item and highlight it if it's currently selected. They're called by the **Draw()** function of the BMenu where the item is located whenever the menu is required to display itself; they don't need to be called from within application code.

However, they can both be overridden by derived classes that display something other than a textual label. The **Draw()** function is called first. It draws the background for the entire item, then calls **DrawContent()** to draw the label within the item's content area. After **DrawContent()** returns, it draws the check mark (if the item is currently marked) and the keyboard shortcut (if any). It finishes by calling **Highlight()** if the item is currently selected.

Both functions draw by calling functions of the BMenu in which the item is located. For example:

```
void MyItem::DrawContent()
{
   . . .
   Menu()->DrawBitmap(image);
   . . .
}
```

A derived class can override either **Draw()**, if it needs to draw the entire item, or **DrawContent()**, if it needs to draw only within the content area. A **Draw()** function can find the frame rectangle

it should draw within by calling the BMenuItem's Frame() function; a DrawContent() function can calculate the content area from the point returned by ContentLocation() and the dimensions provided by GetContentSize().

When DrawContent() is called, the pen is positioned to draw the item's label and the high color is appropriately set. The high color may be a shade of gray, if the item is disabled, or black if it's enabled. If some other distinction is used to distinguish disabled from enabled items, DrawContent() should check the item's current state by calling IsEnabled().

If a derived class implements its own DrawContent() function, but still wants to draw a textual string, it should do so by assigning the string as the BMenuItem's label and calling the inherited version of DrawContent(), not by calling DrawString(). This preserves the BMenuItem's ability to display a trigger character in the string.

See also: Highlight(), Frame(), ContentLocation(), GetContentSize()

Frame()

BRect **Frame**(void) const

Returns the rectangle that frames the entire menu item, in the coordinate system of the BMenu to which the item belongs. If the item hasn't been added to a menu, the return value is indeterminate.

See also: BMenu::AddItem()

GetContentSize()

protected: virtual void **GetContentSize**(float **width*, float **height*)

Writes the size of the item's content area into the variables referred to by *width* and *height*. The content area of an item is the area where its label (or whatever substitutes for the label) is drawn.

A BMenu calls GetContentSize() for each of its items as it arranges them in a column or a row; the function is not called for items in a matrix. The information it provides helps determine where each item is located and the overall size of the menu.

GetContentSize() must report a size that's large enough to display the content of the item (and separate one item from another). By default, it reports an area just large enough to display the item's label. This area is calculated from the label and the BMenu's current font.

If you design a class derived from BMenuItem and implement your own Draw() or DrawContent() function, you should also implement a GetContentSize() function to report how much room will be needed to draw the item's contents.

See also: DrawContent(), ContentLocation()

Highlight()

protected: virtual void **Highlight(**bool *flag***)**

Highlights the menu item when *flag* is `true`, and removes the highlighting when *flag* is `false`. Highlighting simply inverts all the colors in the item's frame rectangle (except for the check mark).

This function is called by the **Draw()** function whenever the item is selected and needs to be drawn in its highlighted state. There's no reason to call it yourself, unless you define your own version of **Draw()**. However, it can be reimplemented in a derived class, if items belonging to that class need to be highlighted in some way other than simple inversion.

See also: **Draw()**

Invoke()

protected: virtual status_t **Invoke(**BMessage **message* = NULL**)**

Augments the BInvoker version of **Invoke()** to ensure that only enabled menu items that are attached to a menu hierarchy can be invoked. This function appropriately marks items when the user invokes them. Before sending a message, it adds "when", "source", and "index" field to it, as explained under the BMenuItem constructor.

See also: **BInvoker::Invoke()**

IsEnabled() see SetEnabled()

isMarked() see SetMarked()

IsSelected()

protected: bool **IsSelected(**void**)** const

Returns `true` if the menu item is currently selected, and `false` if not. Selected items are highlighted.

Label() see SetLabel()

Menu()

BMenu ***Menu(**void**)** const

Returns the menu where the item is located, or `NULL` if the item hasn't yet been added to a menu.

See also: `BMenu::AddItem()`

SetEnabled() , IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**
bool **IsEnabled(**void**)** const

`SetEnabled()` enables the BMenuItem if the *enabled* flag is `true`, disables it if *enabled* is `false`, and updates the item if it's visible on-screen. If the item controls a submenu, this function calls the submenu's `SetEnabled()` virtual function, passing it the same flag. This ensures that the submenu is enabled or disabled as well.

`IsEnabled()` returns `true` if the BMenuItem is enabled, its menu is enabled, and all menus above it in the hierarchy are enabled. It returns `false` if the item is disabled or any objects above it in the menu hierarchy are disabled.

Items and menus are enabled by default.

When using these functions, keep in mind that:

- Disabling a BMenuItem that controls a submenu serves to disable the entire menu hierarchy under the item.
- Passing an argument of `true` to `SetEnabled()` is not sufficient to enable the item if it's located in a disabled branch of the menu hierarchy. It can only undo a previous `SetEnabled()` call (with an argument of `false`) on the same item.

See also: `BMenu::SetEnabled()`

SetLabel() , Label()

virtual void **SetLabel(**const char **string***)**
const char ***Label(**void**)** const

`SetLabel()` frees the item's current label and copies *string* to replace it. If the menu is visible on-screen, it will be redisplayed with the item's new label. If necessary, the menu will become wider (or narrower) so that it fits the new label.

The Interface Kit calls this virtual function to:

- Set the initial label of an item that controls a submenu to the name of the submenu, and
- Subsequently set the item's label to match the marked item in the submenu, if the submenu was set up to have this feature.

Label() returns a pointer to the current label.

See also: BMenu::SetLabelFromMarked(), the BMenuItem constructor

---

SetMarked() , IsMarked()

virtual void **SetMarked(**bool *flag***)**
bool **IsMarked(**void**)** const

SetMarked() adds a check mark to the left of the item label if *flag* is `true`, or removes an existing mark if *flag* is `false`. If the menu is visible on-screen. it's redisplayed with or without the mark.

IsMarked() returns whether the item is currently marked.

See also: BMenu::SetLabelFromMarked(), BMenu::FindMarked()

SetShortcut() , Shortcut()

virtual void **SetShortcut(**char *shortcut*, uint32 *modifiers***)**
char **Shortcut(**uint32 **modifiers* = NULL**)** const

SetShortcut() sets the *shortcut* character that's displayed at the right edge of the menu item and the set of *modifiers* that are associated with the character. These two arguments work just like the arguments passed to the BMenuItem constructor. See the constructor for a more complete description.

Shortcut() returns the character that's used as the keyboard shortcut for invoking the item, and writes a mask of all the modifier keys the shortcut requires to the variable referred to by *modifiers*. Since the Command key is required to operate the keyboard shortcut for any menu item, B_COMMAND_KEY will always be part of the *modifiers* mask. The mask can also be tested against the B_CONTROL_KEY, B_OPTION_KEY, and B_SHIFT_KEY constants.

The shortcut is initially set by the BMenuItem constructor.

SetTrigger() , Trigger()

virtual void **SetTrigger(**char *trigger***)**
char **Trigger(**void**)** const

SetTrigger() sets the *trigger* character that the user can type to invoke the item while the item's menu is open on-screen. If a *trigger* is not set, the Interface Kit will select one for the item, so it's not necessary to call SetTrigger().

The character passed to this function has to match a character displayed in the item—either the keyboard shortcut or a character in the label. The case of the character doesn't matter;

lowercase arguments will match uppercase characters in the item and uppercase arguments will match lowercase characters. When the item can be invoked by its trigger, the trigger character is underlined.

If more than one character in the item matches the character passed, **SetTrigger()** tries first to mark the keyboard shortcut. Failing that, it tries to mark an uppercase letter at the beginning of a word. Failing that, it marks the first instance of the character in the label.

If the *trigger* doesn't match any characters in the item, the item won't have a trigger, not even one selected by the system.

Trigger() returns the character set by **SetTrigger()**, or NULL if **SetTrigger()** didn't succeed or if **SetTrigger()** was never called and the trigger is selected automatically.

See also: **BMenu::SetTriggersEnabled()**

Shortcut() see SetShortcut()

Submenu()

BMenu ***Submenu**(void) const

Returns the BMenu object that the item controls, or NULL if the item doesn't control a submenu.

See also: the BMenuItem constructor, the BMenu class

Trigger() see SetTrigger()

TruncateLabel()

protected: virtual void **TruncateLabel**(float *maxWidth*, char **newLabel*)

Removes characters from the middle of the item label and replaces them with an ellipsis. This is done so that the label will fit within *maxWidth* coordinate units. The shortened string is copied into the *newLabel* buffer.

This function is called by the BMenuItem when it draws the item's label, but only if it's necessary to fit a long item into a smaller space. It can be reimplemented by derived classes to do a better job of shortening the string based on the actual content of the label.

Your version of **TruncateLabel()** should be careful to not cut the trigger character from the string.

See also: **BFont::GetTrunctatedStrings()**

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_label"	`B_STRING_TYPE`	Text label of the menu item.
"_disable"	`B_BOOL_TYPE`	`true` if menu item is disabled.
"_marked"	`B_BOOL_TYPE`	`true` if menu item is marked.
"_user_trig"	`B_INT32_TYPE`	User-defined trigger character.
"_shortcut"	`B_INT32_TYPE`	Shortcut character.
"_mods"	`B_INT32_TYPE`	Shortcut modifiers.
"_msg"	`B_MESSAGE_TYPE`	Model message for the item.
"_submenu"	`B_MESSAGE_TYPE`	Submenu (only in deep copy).

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if no shortcut key was defined, the "_shortcut" and "_mods" fields won't be found in the archive.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuItem Function Summary

Function Summary

Function	Description
ContentLocation()	protected: Returns location of item's content area
Draw()	protected: Draws the menu item
DrawContent()	protected: Draws the item label
Frame()	Returns frame rectangle of menu item
GetContentSize()	protected: Returns size of item's content area
Highlight()	protected: Sets or removes highlight of item
Invoke()	protected: Provided for implementation
IsEnabled()	Tests whether the item is enabled
IsMarked()	Tests whether the item is marked
IsSelected()	protected: Tests whether the item is selected
Label()	Returns item's label
Menu()	Returns menu to which the item is attached
BMenuItem()	Constructor
~BMenuItem()	Destructor
SetEnabled()	Enables or disables item
SetLabel()	Sets item's label
SetMarked()	Marks or unmarks item
SetShortcut()	Sets item's shortcut character

SetTrigger()	Sets item's trigger character
Shortcut()	Returns item's shortcut character
Submenu()	Returns the submenu that the item controls
Trigger()	Returns item's trigger character
TruncateLabel()	protected: Fits item's label to specified width

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPopUpMenu

Derived from: public BMenu
Declared in: be/interface/PopUpMenu.h
Library: libbe.so
Summary: *more...*

A BPopUpMenu is a menu that's typically used in isolation, rather than as part of an extensive menu hierarchy. By default, it operates in radio mode—the last item selected by the user, and only that item, is marked in the menu.

A menu of this kind can be used to choose one from a set of mutually exclusive states—to pick a paper size or paragraph style, for example. It shouldn't be used to group different kinds of choices (as other menus may), nor should it include items that initiate actions rather than set states, except in certain well-defined cases.

A pop-up menu can be used in any of three ways:

- It can be controlled by a BMenuBar object, often one that contains just a single item. The BMenuBar, in effect, functions as a button that pops up a list. The label of the marked item in the list can be displayed as the label of the controlling item in the BMenuBar. In this way, the BMenuBar is able to show the current state of the hidden menu. When this is the case, the menu pops up so its marked item is directly over the controlling item.
- A BPopUpMenu can also be controlled by a view other than a BMenuBar. It might be associated with a particular image the view displays, for example, and appear over the image when the user moves the cursor there and presses the mouse button. Or it might be associated with the view as a whole and come up under the cursor wherever the cursor happens to be. When the view is notified of a mouse-down event, it calls BPopUpMenu's `Go()` function to show the menu on-screen.
- The BPopUpMenu might also be controlled by a particular mouse button, typically the secondary mouse button. When the user presses the button, the menu appears at the location of the cursor. Instead of passing responsibility for the mouse-down event to a BView, the BWindow would intercept it and place the menu on-screen.

Other than `Go()` (and the constructor), this class implements no functions that you ever need to call from application code. In all other respects, a BPopUpMenu can be treated like any other BMenu.

Constructor and Destructor

BPopUpMenu()

```
BPopUpMenu(const char *name, bool radioMode = true,
    bool labelFromMarked = true,
    menu_layout layout = B_ITEMS_IN_COLUMN)
BPopUpMenu(BMessage *archive)
```

Initializes the BPopUpMenu object. If the object is added to a BMenuBar, its *name* also becomes the initial label of its controlling item (just as for other BMenus).

If the *labelFromMarked* flag is `true` (as it is by default), the label of the controlling item will change to reflect the label of the item that the user last selected. In addition, the menu will operate in radio mode (regardless of the value passed as the *radioMode* flag). When the menu pops up, it will position itself so that the marked item appears directly over the controlling item in the BMenuBar.

If *labelFromMarked* is `false`, the menu pops up so that its first item is over the controlling item.

If the *radioMode* flag is `true` (as it is by default), the last item selected by the user will always be marked. In this mode, one and only one item within the menu can be marked at a time. If *radioMode* is `false`, items aren't automatically marked or unmarked.

However, the *radioMode* flag has no effect unless the *labelFromMarked* flag is `false`. As long as *labelFromMarked* is `true`, radio mode will also be `true`.

The *layout* of the items in a BPopUpMenu can be either **`B_ITEMS_IN_ROW`** or the default **`B_ITEMS_IN_COLUMN`**. It should never be **`B_ITEMS_IN_MATRIX`**. The menu is resized so that it exactly fits the items that are added to it.

The new BPopUpMenu is empty; you add items to it by calling BMenu's **`AddItem()`** function.

See also: **`BMenu::SetRadioMode()`**, **`BMenu::SetLabelFromMarked()`**

~BPopUpMenu()

```
virtual ~BPopUpMenu()
```

Does nothing. The BMenu destructor is sufficient to clean up after a BPopUpMenu.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BPopUpMenu object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BPopUpMenu, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `BMenu::Archive()`

Member Functions

Archive()

virtual status_t **Archive(**BMessage ***archive**, bool *deep* = true**)** const

Calls the inherited version of **Archive()** and stores the BPopUpMenu in the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

Go()

BMenuItem ***Go(**BPoint *screenPoint*,
    bool *deliversMessage* = false,
    bool *openAnyway* = false,
    bool *asynchronous* = false**)**
BMenuItem ***Go(**BPoint *screenPoint*,
    bool *deliversMessage*,
    bool *openAnyway*,
    BRect *clickToOpenRect*,
    bool *asynchronous* = false**)**

Places the pop-up menu on-screen so that its left top corner is located at *screenPoint* in the screen coordinate system. If the *asynchronous* flag is `true`, **Go()** returns right away; the return value is `NULL`. Otherwise, it doesn't return until the user dismisses the menu from the screen. If the user invoked an item in the menu, it returns a pointer to the item. If no item was invoked, it returns `NULL`.

Go() is typically called from within the **MouseDown()** function of a BView. For example:

```
void MyView::MouseDown(BPoint point)
{
   BMenuItem *selected;
   BMessage *copy;
   . . .
   ConvertToScreen(&point);
   selected = myPopUp->Go(point);
   . . .
   if ( selected ) {
      BLooper *looper;
      BHandler *target = selected->Target(&looper);
      looper->PostMessage(selected->Message(), target);
   }
```

```
    . . .
}
```

Go() operates in two modes:

- If the *deliversMessage* flag is `true`, the BPopUpMenu works just like a menu that's controlled by a BMenuBar. When the user invokes an item in the menu, the item posts a message to its target.
- If the *deliversMessage* flag is `false`, a message is not posted. Invoking an item doesn't automatically accomplish anything. It's up to the application to look at the returned BMenuItem and decide what to do. It can mimic the behavior of other menus and post the message—as shown in the example above—or it can take some other course of action.

Go() always puts the pop-up menu on-screen, but ordinarily keeps it there only as long as the user holds a mouse button down. When the user releases the button, the menu is hidden and **Go()** returns. However, the *openAnyway* flag and the *clickToOpenRect* arguments can alter this behavior so that the menu will stay open even when the user releases the mouse button (or even if a mouse button was never down). It will take another user action—such as invoking an item in the menu or clicking elsewhere—to dismiss the menu.

If the *openAnyway* flag is `true`, **Go()** keeps the menu on-screen even if no mouse buttons are held down. This permits a user to open and operate a pop-up menu from the keyboard. If *openAnyway* is `false`, mouse actions determine whether the menu stays on-screen.

If the user has the click-to-open menu preference turned on and releases the mouse button while the cursor lies inside the *clickToOpenRect* rectangle, **Go()** interprets the action as clicking to open the menu and keeps it on-screen. If the cursor is outside the rectangle when the mouse button goes up, the menu is removed from the screen and **Go()** returns. The rectangle should be stated in the screen coordinate system.

Once `Go()` returns, your application should delete the BPopUpMenu object.

See also: **`BMenuItem::SetMessage()`**

ScreenLocation()

protected: virtual BPoint **`ScreenLocation(`**void**`)`**

Determines where the pop-up menu should appear on-screen (when it's being run automatically, not by **Go()**). As explained in the description of the class constructor, this largely depends on whether the label of the superitem changes to reflect the item that's currently marked in the menu. The point returned is stated in the screen coordinate system.

This function is called only for BPopUpMenus that have been added to a menu hierarchy (a BMenuBar). You should not call it to determine the point to pass to **Go()**. However, you can

override it to change where a customized pop-up menu defined in a derived class appears on-screen when it's controlled by a BMenuBar.

See also: `BMenu::SetLabelFromMarked()`, `BMenu::ScreenLocation()`, the BPopUpMenu constructor

SetAsyncAutoDestruct() , AsyncAutoDestruct()

void **SetAsyncAutoDestruct**(bool *state*)
bool **AsyncAutoDestruct**(void) const

`SetAsyncAutoDestruct()` lets you specify whether you want BPopUpMenu to automatically delete its BMenu when the pop-up menu closes. If you want the BMenu to be deleted, specify true for state.

`AsyncAutoDestruct()` returns the current state of the flag.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPopUpMenu Function Summary

Function Summary

Function	Description
Archive()	Archives the BPopUpMenu into a BMessage
AsyncAutoDestruct()	Returns whether or not the BMenu will be deleted automatically by an asynchronous menu
Go()	Displays the menu on the screen
Instantiate()	Creates a BPopUpMenu from a BMessage
BPopUpMenu()	Constructor
~BPopUpMenu()	Destructor
ScreenLocation()	protected: Hook to position menu
SetAsyncAutoDestruct()	Selects whether or not the BMenu should be deleted automatically by an asynchronous menu

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuBar Function Summary

Function Summary

Function	Description
Archive()	Archives a BMenuBar into a BMessage
AttachedToWindow()	Provided for implementation
Border()	Returns menu border
Draw()	Provided for implementation
Hide()	Provided for implementation
Instantiate()	Creates a BMenuBar from a BMessage
BMenuBar()	Constructor
~BMenuBar()	Destructor
MouseDown()	Provided for implementation
SetBorder()	Sets menu border
Show()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPropertyInfo

Derived from: BFlattenable
Declared in: be/app/PropertyInfo.h
Library: libbe.so
Summary: *more...*

BPropertyInfo is a simple class that manages scripting. A program describes its scripting interface to a BPropertyInfo object through an array of property_info structures, with each entry describing a piece of the scripting suite. The structure definition:

```
struct property_info {
char *name;
uint32 commands[10];
uint32 specifiers[10];
char *usage;
uint32 extra_data;
};
```

- *name* provides the name of the property this structure describes.
- *commands* is a zero-terminated array of commands understood by the property, i.e. `B_GET_PROPERTY`. If the first element is 0, it represents a wildcard matching all possible commands.
- *specifiers* is a zero-terminated array of the specifiers understood by the property, i.e. `B_DIRECT_SPECIFIER`. If the first element is 0, it represents a wildcard matching all possible specifiers.
- *usage* gives a human-readable string describing the property and its allowable commands and specifiers.
- *extra_data* is an area free for general use; the operating system does not touch its contents.

A BPropertyInfo is instantiated by passing a zero-terminated array of property_info to its constructor. A typical initialization of BPropertyInfo looks like:

```
static property_info prop_list[] = {
{ "duck", {B_GET_PROPERTY, B_SET_PROPERTY, 0},
{B_DIRECT_SPECIFIER, B_INDEX_SPECIFIER, 0}, "get or set duck"},
{ "head", {B_GET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "get head"},
{ "head", {B_SET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "set head"},
{ "feet", {0}, {0}, "can do anything with his orange feet"},
0 // terminate list
```

```
};

BPropertyInfo prop_info(prop_list);
```

Since BPropertyInfo only stores a pointer to the array, it is important that the life span of the array is at least as long as that of the BPropertyInfo object.

Notice that BPropertyInfo doesn't impose any particular structure upon the array; in particular, not all commands and specifiers for a given property need be placed in a single entry in the array. You are free to organize your scripting suite in whatever manner is most convenient for your particular object.

BPropertyInfo is a descendant of BFlattenable, and can therefore be used to store a description of an object's supported scripting suite. This is particularly useful when overriding **GetSupportedSuites()**:

```
status_t MyHandler::GetSupportedSuites(BMessage *msg)
{
msg->AddString("suites", "suite/vnd.Me-my_handler");
BPropertyInfo prop_info(prop_list);
msg->AddFlat("messages", &prop_info);
return baseClass::GetSupportedSuites(msg);
}
```

Naturally, BPropertyInfo is equally as useful in interpreting the results obtained from querying an object for its supported suites.

BPropertyInfo defines the **FindMatch()** method designed to simplify the implementation of **ResolveSpecifier()**. It returns the index of the property info matching the description given to it, or -1 if none match. This reduces **ResolveSpecifier()** in the simplest cases to:

```
BHandler *MyHandler::ResolveSpecifier(BMessage *msg, int32 index,
BMessage *spec, int32 form, const char *prop)
{
BPropertyInfo prop_info(prop_list);
if (prop_info.FindMatch(msg, index, spec, form, prop) >= 0)
return this;
return baseClass::ResolveSpecifier(msg, index, spec, form, prop);
}
```

Of course, for more complicated objects, **ResolveSpecifier()** may need to set the target handler to an object other than itself, so more processing may be required. In those cases, the object can use the index returned by **FindMatch()** to help it determine the target of the scripting message.

Constructor and Destructor

BPropertyInfo()

```
BPropertyInfo(property_info *p = NULL, bool free_on_delete = false)
```

Initializes the object with the specified zero-terminated array *p* of property_info. Passing `true` in *free_on_delete* instructs the object to free the memory associated with the property_info when the object is destroyed. BPropertyInfo does not copy the array, so it is important that the array is not deleted or otherwise destroyed while the BPropertyInfo is in use.

~BPropertyInfo()

```
~BPropertyInfo()
```

If *free_on_delete* set to `true` in the constructor, the destructor frees all memory associated with the property_info. Otherwise, does nothing.

Member Functions

AllowsTypeCode()

Implementation detail. See **`BFlattenable::AllowsTypeCode()`**.

FindMatch()

int32 **FindMatch(**BMessage **msg*, int32 *index*, BMessage **spec*, int32 *form*,
const char **prop*, void **data* = NULL**)** const

Passed a property name in *prop*, a specifier in *form*, and a command in *msg->what*, searches the property_info array for an item supporting the specified scripting request. If *index* is nonzero, then `FindMatch()` only searches those property_info structures with the wildcard command (first element of command array equal to 0). Otherwise, it searches through all available property_info structures for a match. If a match is found, it fills the memory at *data* with the contents of the extra_data field of the match and returns the index of the match in the array. Otherwise, it returns `B_ERROR`.

Flatten

Implementation detail. See **`BFlattenable::Flatten()`**.

FlattenedSize()

Implementation detail. See **`BFlattenable::FlattenedSize()`**.

IsFixedSize()

Implementation detail. See **`BFlattenable::IsFixedSize()`**.

TypeCode()

Implementation detail. See **`BFlattenable::TypeCode()`**.

PrintToStream()

void **PrintToStream(**void**)** const

Prints information about the BPropertyInfo to standard output.

PropertyInfo()

```
const property_info *PropertyInfo(void) const
```

Returns the property_info list associated with the object.

Unflatten()

Implementation detail. See `BFlattenable::Unflatten()`.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPropertyInfo Function Summary

Function Summary

Function	Description
AllowsTypeCode()	Provided for implementation
FindMatch()	Passed a scripting message, returns a matching property_info
FlattenedSize()	Provided for implementation
IsFixedSize()	Provided for implementation
PrintToStream()	Prints information about the object to standard output
PropertyInfo()	Returns the property_info list associated with the object
~BPropertyInfo()	Destructor
TypeCode()	Provided for implementation
Unflatten()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

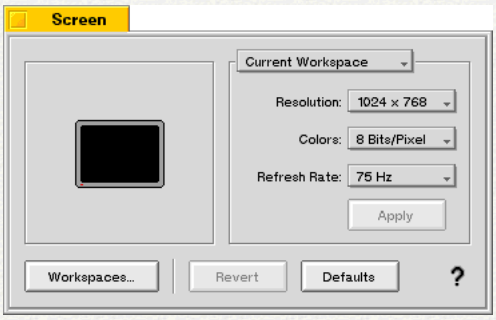
BBox

Derived from: public BView
Declared in: be/interface/Box.h
Library: libbe.so
Summary: *more...*

A BBox is a purely cosmetic and organizational view that draws a labeled border around its (view) children. The border and label are drawn within the BBox's frame rectangle.

- The current pen size of the view determines the width of the border; by default it's 1.0 coordinate unit.
- The label is drawn in the current font, which by default is the system bold font.
- The label is drawn in the current high color; the default high color is black.

The BBox's label can be a string or it can be another view, such as a control. For example, a popup menu might be used as the BBox's label, to let the user select a group of options to view in a preference panel:



Constructor and Destructor

BBox()

```
BBox(BRect frame, const char *name = NULL,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_FRAME_EVENTS |
        B_NAVIGABLE_JUMP,
        border_style border = B_FANCY_BORDER)
BBox(BMessage *archive)
```

Initializes the BBox by passing the *frame*, *name*, *resizingMode*, and *flags* to the BView constructor, and sets the style of its border to *border*. Three border styles are possible:

B_PLAIN_BORDER	The border is a simple line, lighter on the left and top than on the right and bottom so that the box looks raised from the surrounding surface.
B_FANCY_BORDER	The border is a fancier line that looks like a 3D groove inset into the surrounding surface of the view.
B_NO_BORDER	There is no border. This option is not that useful for a BBox object; it turns the box into something other than a box.

The constructor also sets the font for displaying the BBox's label to the system bold font (**be_bold_font**). However, the new object doesn't have a label; call **SetLabel()** to assign it one.

See also: **SetLabel()**

~BBox()

```
virtual ~BBox()
```

Frees the label, if the BBox has one.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow**(void)

Makes the BBox's background view color and its low color match the background color of its new parent.

Border() see SetBorder()

Draw()

virtual void **Draw**(BRect *updateRect*)

Draws the box and its label. This function is called automatically in response to update messages.

FrameResized()

virtual void **FrameResized**(float *width*, float *height*)

Makes sure that the parts of the box that change when it's resized are redrawn.

See also: `BView::FrameResized()`

Label() see SetLabel()

SetBorder() , Border()

virtual void **SetBorder**(border_style *border*)
border_style **Border**(void) const

These functions set and return the style of border the BBox draws—`B_PLAIN_BORDER`, `B_FANCY_BORDER`, or `B_NO_BORDER`. The border style is initially set by the BBox constructor.

SetLabel() , Label() , LabelView()

void **SetLabel**(const char **string*)
status_t **SetLabel**(BView **viewLabel*)

const char ***Label**(void) const
BView ***LabelView**(void) const

These functions set and return the label that's displayed along the top edge of the box. **`SetLabel()`** copies *string* and makes it the BBox's label, freeing the previous label, if any. If *string* is `NULL`, it removes the current label and frees it.

Alternately, you can use the second form of `SetLabel()` to specify a view to be used as the BBox's label. This view can be anything, including controls.

`Label()` returns a pointer to the BBox's current label, or `NULL` if it doesn't have one (or the label is a view).

`LabelView()` returns a pointer to the BView that's being used as the BBox's label, or **`NULL`** if either there is no label, or the label is a string.

RETURN CODES

- B_OK. No error occurred.
- No other errors are returned.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_label"	`B_STRING_TYPE`	The box's descriptive text.
"_style"	`B_BOOL_TYPE`	Exists and is `true` if the style is not `B_FANCY_BORDER`.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBox Function Summary

Function Summary

Function	Description
AttachedToWindow()	Provided for implementation
Border()	Returns the border style
BBox()	Constructor
~BBox()	Destructor
Draw()	Provided for implementation
FrameResized()	Provided for implementation
Label()	Sets the view label
LabelView()	Returns the text label
SetBorder()	Sets the border style
SetLabel()	Sets the text label

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Functions

Library: libbe.so
Summary: *more...*

This section describes the global C functions defined in the Interface Kit. All these functions deal with aspects of the systemwide environment for the user interface—the screen, workspaces, installed fonts, the list of possible colors, and various user preferences. For mouse and keyboard functions, see "Input Functions".

In almost all cases, you need a valid (but not necessarily running) **be_app** object in order to call these functions.

Functions

activate_workspace() , current_workspace()

Declared in: be/interface/InterfaceDefs.h

void **activate_workspace(**int32 *workspace***)**
int32 **current_workspace(**void**)**

These functions set and return the active workspace, the one that's currently displayed on-screen. The workspace is identified by an index, 0-based.

See also: **BWindow::WorkspaceActivated()**

bitmaps_support_space()

Declared in: be/interface/GraphicsDefs.h

bool **bitmaps_support_space(color_space** *space*, uint32 **supportFlags***)*

Returns **true** if BBitmaps can contain graphics in the specified color space, or **false** if not.

The uint32 pointed to by *supportFlags* will be set to a bit field of flags, further describing the support for the specified color space by the BBitmap class. The flags are:

B_VIEWS_SUPPORT_DRAW_BITMAP	**BView::DrawBitmap()** supports drawing BBitmaps of the specified color space.
B_BITMAPS_SUPPORT_ATTACHED_VIEWS	Indicates that BBitmaps in the specified color space support attached BViews.

count_font_families() see get_font_family()
count_font_styles() see get_font_style()
count_workspaces() see set_workspace_count()
current_workspace() see activate_workspace()
get_click_speed() see set_click_speed()

get_deskbar_frame()

Declared in: be/interface/InterfaceDefs.h

status_t **get_deskbar_frame(**BRect **frame***)*

Writes the screen coordinates of the current frame of the Deskbar window into *frame*. Returns `B_OK` on success or an appropriate error code on failure.

---

get_font_family() , count_font_families() , get_font_style() , count_font_styles()

Declared in: be/interface/Font.h

status_t **get_font_family(**int32 *index*, font_family **family*, uint32 **flags* = NULL**)**
int32 **count_font_families(**void**)**
status_t **get_font_style(**font_family *family*, int32 *index*, font_style **style*, uint32 **flags* = NULL**)**
int32 **count_font_styles(**font_family *family***)**

These functions are used in combination to get the names of the families and styles of all installed fonts. For example:

```
int32 numFamilies = count_font_families();
for ( int32 i = 0; i < numFamilies; i++ ) {
   font_family family;
   uint32 flags;
   if ( get_font_family(i, &family, &flags) == B_OK ) {
      . . .
      int32 numStyles = count_font_styles(family);
      for ( int32 j = 0; j < numStyles; j++ ) {
         font_style style;
         if ( get_font_style(family, j, &style, &flags) == B_OK ) {
            . . .
         }
      }
   }
}
```

`get_font_family()` reads one family name from the list of installed fonts, the name at *index*, and copies it into the *family* buffer; **`count_font_families()`** returns the number of font families currently installed. Similarly, **`get_font_style()`** reads the name of the style at *index* and copies into the *style* buffer. Since each family can have a different set of styles, a *family* name must be passed to **`get_font_style()`**; **`count_font_styles()`** returns the number of styles for the particular *family*. Family and style names can be up to 64 bytes long including a null terminator. Indices begin at 0.

The names of installed font families and styles are not indexed in any particular order. You might want to alphabetize them before displaying them to the user in a menu or list.

If you pass a *flags* argument to **get_font_family()** and **get_font_style()**, they will place a mask with useful information about the particular family or style in the variable that the argument refers to. Currently there are just two flags:

B_IS_FIXED	Indicates that the font is a nonproportional, or fixed-width, font—one for which all characters have the same width.
B_HAS_TUNED_FONT	Indicates that the family or style has versions of the font especially adapted or "tuned" for on-screen display.

If neither flag applies, the variable that *flags* points to will be set to 0.

If you find a family and style that has a tuned font, you can set a BFont object to that family and style, then call the object's **GetTunedInfo()** function to get details about exactly which combination of font properties (for example, which font sizes) have tuned counterparts. If you set a BFont so that it has those properties and make it a BView's current font, the tuned version will be used when the BView draws to the screen.

It's possible for the user to install or remove fonts while the application is running. However, unless **update_font_families()** has been called to get the updated list, **get_font_family()** will provide information on the same set of fonts each time it's called. The list isn't automatically updated.

See also: **update_font_families()**, **BView::SetFont()**, **BFont::SetFamilyAndStyle()**

- get_key_repeat_delay() see set_key_repeat_rate()
- get_key_repeat_rate() see set_key_repeat_rate()
- see get_key_info()
- see get_key_map()
- get_menu_info() see set_menu_info()

- get_pixel_size_for()

Declared in: be/interface/GraphicsDefs.h

status_t **get_pixel_size_for(color_space** *space*, size_t **pixelChunk*,
size_t **rowAlignment*, size_t **pixelsPerChunk***)*

Given the specified color space, returns information about pixel and row alignment for that color space.

On return, *pixelChunk* indicates

- get_mouse_map() see set_mouse_map()
- get_mouse_speed() see set_mouse_map()

- get_mouse_type() see set_mouse_map()
- set_scroll_bar_info()

---

idle_time()

Declared in: be/interface/InterfaceDefs.h

bigtime_t **idle_time**(void) const

Returns the number of microseconds since the user last manipulated the mouse or keyboard. This information isn't specific to a particular application; **idle_time()**set_scroll_bar_info() tells you when the user last directed an action at *any* application, not just yours.

---

keyboard_navigation_color()

Declared in: be/interface/InterfaceDefs.h

rgb_color **keyboard_navigation_color**(void)

Returns the color that should be used to mark the BView that's currently in focus, when the user can change the focus from the keyboard. The keyboard navigation color is typically used to underline the labels of control devices and to outline text fields where the user can type.

See also: the BView class

- see modifiers()

---

run_add_printer_panel() , run_select_printer_panel()

Declared in: be/interface/InterfaceDefs.h

void **run_add_printer_panel**(void)
void **run_select_printer_panel**(void)

These two functions have the Print Server place panels on-screen where the user can set up a printer and choose which printer to use. **run_add_printer_panel()** displays a panel that informs the server about a new printer. **run_select_printer_panel()** displays a panel that lists all known printers and lets the user select one.

See also: the BPrintJob class

- see set_click_speed()
- see set_key_repeat_rate()

- see set_keyboard_locks()

---

set_menu_info() , get_menu_info()

Declared in: be/interface/Menu.h

status_t **set_menu_info(menu_info** *info**)**
status_t **get_menu_info(menu_info** *info**)**

These functions set and get the user's preferences for how menus should look and work. User's express their preferences with the Menu application, which calls **set_menu_info()**. **get_menu_info()** writes the current preferences into the **menu_info** structure that into refers to. This structure contains the following fields:

float **font_size**
The size of the font that will be used to display menu items.

font_name font
The name of the font that's used to display menu items.

rgb_color background_color
The background color of the menu.

int32 **separator**
The style of horizontal line that separates groups of items in a menu. The value is an index ranging from 0 through 2; there are three possible separators.

bool **click_to_open**
Whether it's possible to open a menu by clicking in the item that controls it. The default value is `true`.

bool **triggers_always_shown**
Whether trigger characters are always marked in menus and menu bars, regardless of whether the menu hierarchy is the target for keyboard actions. The default value is `false`.

(At present, both functions always return **B_OK**.)

See also: the BMenu class

- see set_modifier_key()
- see set_mouse_map()

---

set_screen_space()

Declared in: be/interface/InterfaceDefs.h

status_t **set_screen_space(**int32 *index*, uint32 *space*, bool *makeDefault* = true**)**

Changes the configuration of the screen—its depth and dimensions—to match the values specified in the *space* constant, which can be any of the following:

B_8_BIT_640x400			
B_8_BIT_640x480	**B_15_BIT_640x480**	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_15_BIT_800x600**	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_15_BIT_1024x768**	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_15_BIT_1152x900**	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_15_BIT_1280x1024**	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_15_BIT_1600x1200**	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

The first part of the constant designates the screen depth and color space. `B_8_BIT`... refers to the **B_CMAP8** color space. The other values correspond to the **B_RGB15**, **B_RGB16**, and **B_RGB32** color spaces, as appropriate. Although constants are defined for 15-bit screen depths, the operating system currently doesn't support them. The second part of the constant designates the pixel resolution of the screen. For example, **B_32_BIT_1280x1024** means that the frame buffer is 32 bits deep (**B_RGB32**) while the screen grid is 1,280 pixels wide and 1,024 pixels high.

This function affects the screen at *index*. Since the BeOS currently doesn't support more than one screen, the only index that makes sense is 0.

The change to the screen takes effect immediately. If the *makeDefault* flag is `true`, the new configuration also becomes the default and will be used the next time the machine is turned on. If *makeDefault* is `false`, the configuration is in effect for the current session only.

Since not all configurations are possible for all graphics cards, **set_screen_space()** can fail. It returns **B_OK** if successful, and **B_ERROR** if not.

This function is designed for preferences applications—like the Screen application—that permit users to make system-wide choices about the screen. Other applications should respect those choices and refrain from modifying them.

The current screen configuration can be obtained from the BScreen object.

See also: the BScreen class, **BWindow::ScreenChanged()**, **The Game Kit** chapter

---

set_scroll_bar_info() , get_scroll_bar_info()

Declared in: be/interface/InterfaceDefs.h

status_t **set_scroll_bar_info(**scroll_bar_info **info***)**

status_t **get_scroll_bar_info(**scroll_bar_info **info***)**

These functions set and report preferences that the BScrollBar class uses when it creates a new scroll bar. **set_scroll_bar_info()** reads the values contained in the **scroll_bar_info**

structure that *info* refers to and sets the system-wide preferences accordingly; `get_scroll_bar_info()` writes the current preferences into the structure provided.

The `scroll_bar_info` structure contains the following fields:

bool **`proportional`**
: **`true`** if scroll bars should have a knob that grows and shrinks to show what proportion of the document is currently visible on-screen, and `false` if not. Scroll knobs are proportional by default.

bool **`double_arrows`**
: **`true`** if a set of double arrows (for scrolling in both directions) should appear at each end of the scroll bar, or `false` if only single arrows (for scrolling in one direction only) should be used. Double arrows are the default.

int32 **`knob`**
: An index that picks the pattern for the knob. Only values of 0, 1, and 2 are currently valid. The patterns can be seen in the ScrollBar preferences application. The pattern at index 1 is the default.

int32 **`min_knob_size`**
: The length of the scroll knob, in pixels. This is the minimum size for a proportional knob and the fixed size for one that's not proportional. The default is 15.

The user can set these preferences with the ScrollBar application. Applications can call `get_scroll_bar_info()` to find out what choices the user made, but should refrain from calling `set_scroll_bar_info()`. That function is desigined for utilities, like the ScrollBar application, that enable users to set preferences that are respected system-wide.

If successful, these functions return `B_OK`; if not, they return `B_ERROR`.

See also: the BScrollBar class

---

set_workspace_count() , count_workspaces()

Declared in: be/interface/InterfaceDefs.h

void **`set_workspace_count`**(int32 *numWorkspaces*)
int32 **`count_workspaces`**(void)

These functions set and return the number of workspaces the user has available. There can be as many as 32 workspaces and as few as 1. The choice of how many there should be is usually left to the user and the Workspaces application.

See also: `activate_workspace()`

---

system_colors()

Declared in: be/interface/InterfaceDefs.h

const color_map ***system_colors**(void)

Returns a pointer to the system color map. This function duplicates the BScreen **ColorMap()** function, but it permits software that isn't concerned about the on-screen display to get the color map without referring to a particular screen. (Actually it returns the color map for the main screen.)

The **color_map** structure returned by this function belongs to the operating system.

See also: **BScreen::ColorMap()**

---

update_font_families()

Declared in: be/interface/Font.h

bool **update_font_families**(bool *checkOnly*)

Updates the list of installed fonts, so that it reflects any that have been added or removed since the last time the list was updated. Until the list is updated, **get_font_family()** operates assuming the set of fonts that were installed when the application started up. If the list is unchanged since the last update (or since startup), this function returns `false`; if a font has been installed or an installed font has been removed, it returns `true`.

If the *checkOnly* flag is `true`, **get_font_family()** only reports whether the list has changed; it doesn't modify the current list. If the flag is `false`, it contacts the Application Server to get the updated list, a much more expensive operation.

See also: **get_font_family()**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Functions Function Summary

Function Summary

Function	Description
activate_workspace()	Sets the active workspace
bitmaps_support_space()	Indicates whether BBitmaps support a specified color space
count_font_families()	Returns the number of installed font families
count_font_styles()	Returns the number of installed font styles for a given family
count_workspaces()	Returns the number of available workspaces
current_workspace()	Returns the active workspace
get_deskbar_frame()	Returns the Deskbar window frame
get_font_family()	Returns information about a specific font family
get_font_style()	Returns information about a specific font style for a given family
get_menu_info()	Returns the menu preferences
get_pixel_size_for()	Returns information about pixel and row alignment for a color space
get_scroll_bar_info()	Returns the BScrollBar

	preferences
idle_time()	Returns the number of microseconds since the last user action
keyboard_navigation_color()	Returns the color used to mark the focused BView
run_add_printer_panel()	Brings up the add printer dialog
run_select_printer_panel()	Brings up the select printer dialog
set_menu_info()	Sets the menu preferences
set_screen_space()	Sets the screen depth and dimensions
set_scroll_bar_info()	Sets the BScrollBar preferences
set_workspace_count()	Sets the number of available workspaces
system_colors()	Returns a pointer to the system color map
update_font_families()	Forces an update of the list of installed fonts

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPrintJob

Derived from: *none*
Declared in: be/interface/PrintJob.h
Library: libbe.so
Summary: *more...*

A BPrintJob object runs a printing session. It negotiates everything after the user's initial request to print—from engaging the Print Server to calling upon BViews to draw and spooling the results to the printer. It also handles a secondary and somewhat separate matter related to printing—configuring the page layout.

Setting Up the Page Layout

Users typically don't decide how a document fits on a page— the size of the paper, the width of the margins, the orientation of the image, and so on—each time they print. These decisions are usually made when setting up the document, perhaps from a Page Layout menu item, rather than Print.

To set up the page parameters for a document, an application should create a BPrintJob object, assign it a name, and call **ConfigPage()**:

```
status_t MyDocumentManager::SetUpPage()
{
   BPrintJob job("document");
   return job.ConfigPage();
}
```

ConfigPage() has the Print Server interact with the user to set up the page configuration. Configuration settings are stored in a BMessage object that will be handed to the server when the document is printed. The BMessage is important to the server, but the application doesn't need to look at it directly; functions are provided to access the useful data it contains. However, you may want to get the object and store it with the document so that the configuration can be reused whenever the document is printed—and so that the user's previous choices can be the default settings when **ConfigPage()** is called again. This is good behavior for an application to follow, and is highly recommended.

Settings() returns the page configuration the user set up; **SetSettings()** initializes the configuration that's presented to the user. For example:

```
BMessage *setup;
. . .
status_t MyDocumentManager::SetUpPage() {
   status_t result;
   BPrintJob job("document's name");
```

```
   if (setup) {
      job.SetSettings(new BMessage(*setup));
   }

   result = job.ConfigPage();

   if (result == B_OK) {
      setup = job.Settings();

      /* record the settings for your own use */
      paper_rect = job.PaperRect();
      printable_rect = job.PrintableRect();
   }

   return result;
}
```

In this example, the `setup` BMessage presumably is flattened and saved with the document whenever the document is saved, and unflattened whenever the document is open and the page settings are needed.

Printing

To print a document, an application must go through several ordered steps:

- Engaging the Print Server and setting parameters for the job.
- Setting up a spool file to hold image data.
- Asking BViews to draw each page.
- After each page is drawn, putting the data for the page in the spool file.
- Committing the spool file to the Print Server.

A BPrintJob object has member functions that assist with each step.

Setting Up a Print Job

A print job begins when the user requests the application to print something. In response, the application should create a BPrintJob object, assign the job a name, and call **`ConfigJob()`** to initialize the printing environment. For example:

```
BMessage *setup;
. . .
status_t MyDocumentManager::Print()
{
   BPrintJob job("document");
```

```
   status_t err;

   if ( setup )
      job.SetSettings(new BMessage(*setup));
   if ( (err = job.ConfigJob()) == B_OK ) {
      delete setup;
      setup = job.Settings();
   }
   . . .
}
```

So far, this looks much like the code for configuring the page presented in the previous "Setting Up the Page Layout" section. The idea is the same. **ConfigJob()** gets the Print Server ready for a new printing session and has it interact with the user to set up the parameters for the job—which pages, how many copies, and so on. It uses the same settings BMessage to record the user's choices as **ConfigPage()** did, though it records information that's more immediate to the particular printing session.

Again, you may want to store the user's choices with the document so that they can be used to set the initial configuration for the job when the document is next printed. By calling **Settings()**, you can get the job configuration the user set up; **SetSettings()** initializes the configuration that's presented to the user.

Information about the page layout will be required while printing. If that information isn't available in the **Settings()** BMessage, **ConfigJob()** will begin, in essence, by calling **ConfigPage()** so that the server can ask the user to supply it.

To discover which pages the user wants to print, you can call the **FirstPage()** and **LastPage()** functions after **ConfigJob()** returns:

```
int32 pageCount = job.LastPage() - job.FirstPage() + 1;
```

The Spool File

The next step after configuring the job is to call **BeginJob()** to set up a spool file and begin the production of pages. After all the pages are produced, **CommitJob()** is called to commit them to the printer.

```
job.BeginJob();
/* draw pages here*/
job.CommitJob();
```

BeginJob() and **CommitJob()** bracket all the drawing that's done during the job.

Cancellation

A number of things can happen to derail a print job after it has started—most significantly, the user can cancel it at any time. To be sure that the job hasn't been canceled or something

else hasn't happened to defeat it, you can call **CanContinue()** at critical junctures in your code. This function will tell you whether it's sensible to continue with the job. In the following example, a while loop is used to loop through all the pages in the document, or until **CanContinue()** returns false, indicating that the job has been canceled:

```
job.BeginJob();
while (job.CanContinue() && page <= pageCount) {
   /* draw each page here*/
   page++;
}
job.CommitJob();
```

Drawing on the Page

A page is produced by asking one or more BViews to draw within a rectangle that can be mapped to a sheet of paper (excluding the margins at the edge of the paper). **DrawView()** requests one BView to draw some portion of its data and specifies where the data should appear on the page. You can call **DrawView()** any number of times for a single page to ask any number of BViews to contribute to the page. After all views have drawn, **SpoolPage()** spools the data to the file that will eventually be committed to the printer. **SpoolPage()** is called just once for each page. For example:

```
for (int i = job.FirstPage(); job.CanContinue() && i <= job.LastPage(); i++) {
   . . .
   job.DrawView(someView, viewRect, pointOnPage);
   job.DrawView(anotherView, anotherRect, differentPoint);
   . . .
   job.SpoolPage();
}
```

DrawView() calls the BView's **Draw()** function. That function must be prepared to draw either for the screen or on the printed page. It can test the destination of its output by calling the BView **IsPrinting()** function.

Don't use the **BView::Bounds()** function to determine the area to render. Instead, use the update rectangle passed to the **BView::Draw()** function.

A Complete Printing Function

This function puts together all the above code snippets and handles the printing of a document (minus the actual drawing and visual status information presented to the user as printing goes on).

```
status_t MyDocumentManager::Print() {
   status_t result = B_OK;
   BPrintJob job("document's name");
```

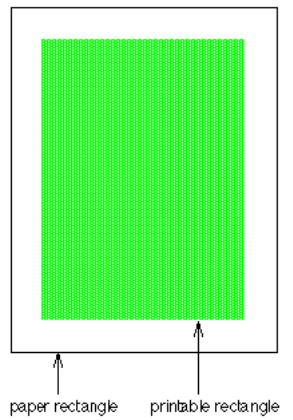
```
// If we have no setup message, we should call ConfigPage()
// You must use the same instance of the BPrintJob object
// (you can't call the previous "PageSetup()" function, since it
// creates its own BPrintJob object).

if (!setup) {
   result = job.ConfigPage();
   if (result == B_OK) {
      // Get the user Settings
      setup = job.Settings();

      // Use the new settings for your internal use
      paper_rect = job.PaperRect();
      printable_rect = job.PrintableRect();
   }
}

if (result == B_OK) {
   // Setup the driver with user settings
   job.SetSettings(setup);

   result = job.ConfigJob();
   if (result == B_OK) {
      // WARNING: here, setup CANNOT be NULL.
      if (setup == NULL) {
         // something's wrong, handle the error and bail out
      }
      delete setup;

      // Get the user Settings
      setup = job.Settings();

      // Use the new settings for your internal use
      // They may have changed since the last time you read it
      paper_rect = job.PaperRect();
      printable_rect = job.PrintableRect();

      // Now you have to calculate the number of pages
      // (note: page are zero-based)
      int32 firstPage = job.FirstPage();
      int32 lastPage = job.LastPage();

      // Verify the range is correct
      // 0 ... LONG_MAX -> Print all the document
      // n ... LONG_MAX -> Print from page n to the end
      // n ... m -> Print from page n to page m
```

```
         if (lastPage > your_document_last_page)
            last_page = your_document_last_page;

         int32 nbPages = lastPage - firstPage + 1;

         // Verify the range is correct
         if (nbPages <= 0)
            return B_ERROR;

         // Now you can print the page
         job.BeginJob();

         // Print all pages
         bool can_continue = job.CanContinue();

         for (int i=firstPage ; can_continue && i<=lastPage ; i++) {
            // Draw all the needed views
            job.DrawView(someView, viewRect, pointOnPage);
            job.DrawView(anotherView, anotherRect, differentPoint);

            // If the document have a lot of pages, you can update a BStatusBar, here
            // or else what you want...
            update_status_bar(i-firstPage, nbPages);

            // Spool the page
            job.SpoolPage();

            // Cancel the job if needed.
            // You can for exemple verify if the user pressed the ESC key
            // or (SHIFT + '.')...
            if (user_has_canceled)
            {
               // tell the print_server to cancel the printjob
               job.CancelJob();
               can_continue = false;
               break;
            }

            // Verify that there was no error (disk full for
exemple)
            can_continue = job.CanContinue();
         }

         // Now, you just have to commit the job!
         if (can_continue)
            job.CommitJob();
         else
            result = B_ERROR;
```

```
        }
    }

    return result;
}
```

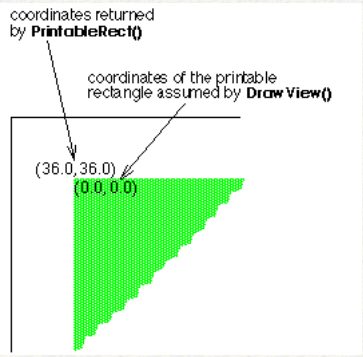
Drawing Coordinates

When a BView draws for the printer, it draws within the *printable rectangle* of a page—a rectangle that matches the size of a sheet of paper minus the unprinted margin around the paper's edge. The **PaperRect()** function returns a rectangle that measures a sheet of paper and **PrintableRect()** returns the printable rectangle, as illustrated in this diagram.



Both rectangles are stated in a coordinate system that has its origin at the left top corner of the page. Thus, the left and top sides of the rectangle returned by **PaperRect()** are always 0.0. **PrintableRect()** locates the printable rectangle on the paper rectangle. However, **DrawView()** assumes coordinates that are local to the printable rectangle—that is, an origin at the left top corner of the printable rectangle rather than the paper rectangle.

The diagram below shows the left top coordinates of the printable rectangle as **PrintableRect()** would report them and as **DrawView()** would assume them, given a half-inch margin.



Draw() always draws in the BView's own coordinate system. Those coordinates are mapped to locations in the printable rectangle as specified by the arguments passed to **DrawView()**.

See also: `BView::IsPrinting()`

Constructor and Destructor

BPrintJob()

```
BPrintJob(const char *name)
```

Initializes the BPrintJob object and assigns the job a *name*. The Print Server isn't contacted until `ConfigPage()` or `ConfigJob()` is called. The spool file isn't created until `BeginJob()` prepares for the production of pages.

See also: `ConfigJob()`, `BeginJob()`

~BPrintJob()

```
virtual ~BPrintJob()
```

Frees all memory allocated by the object.

Member Functions

BeginJob()

```
void BeginJob(void)
```

Opens a spool file for the job and prepares for the production of a series of pages. Call this function only once per printing session—just after initializing the job and just before drawing the first page.

See also: `CommitJob()`, "The Spool File" in the class overview

CancelJob()

```
void CancelJob(void)
```

Cancels the print job programmatically and gets rid of the spool file. The job cannot be restarted; you must destroy the BPrintJob object. Create a new object to renew printing.

CanContinue()

```
bool CanContinue(void)
```

Returns `true` if there's no impediment to continuing with the print job, and `false` if the user has canceled the job, the spool file has grown too big, or something else has happened to terminate printing. It's a good idea to liberally sprinkle `CanContinue()` queries throughout your printing code to make sure that the work you're about to do won't be wasted.

See also: "Cancellation" in the class overview

CommitJob()

```
void CommitJob(void)
```

Commits all spooled pages to the printer. This ends the print job; when `CommitJob()` returns, the BPrintJob object can be deleted. `CommitJob()` can be called only once per job.

See also: `BeginJob()`, "The Spool File" in the class overview

ConfigPage() , ConfigJob()

```
status_t ConfigPage(void)
```

status_t **ConfigJob**(void)

These functions contact the Print Server and have the server interact with the user to lay out the document on a page (in the case of `ConfigPage()`) or to define a print job (in the case of `ConfigJob()`). The page layout includes such things as the orientation of the image (portrait or landscape), the dimensions of the paper on which the document will be printed, and the size of the margins. The job definition includes such things as which pages are to be printed and the number of copies.

Both functions record the user's choices in a BMessage object that `Settings()` returns.

If `SetSettings()` has been called to establish a default configuration for the page layout or the job, these functions will pass it to the Print Server so the server can present it to the user. Otherwise, the server will choose a default configuration to show the user.

These two functions return `status_t` error codes, despite having return values that are declared `int32`. They return `B_ERROR` if they have trouble communicating with the server or if the job can't be established for any reason. They return `B_OK` if all goes well.

See also: `SetSettings()`, "Setting Up the Page Layout" and "Setting Up a Print Job" in the class overview

---

DrawView() , SpoolPage()

virtual void **DrawView**(BView **view*, BRect *rect*, BPoint *point*)

void **SpoolPage**(void)

`DrawView()` calls upon a *view* to draw the *rect* portion of its display at the location specified by *point* on the page. As a result, the *view*'s `Draw()` function will be called with *rect* passed as the update rectangle. The rectangle should be stated in the BView's coordinate system. The *point* should be stated in a coordinate system that has the origin at the top left corner of the printable rectangle. Together the *rect* and *point* should be fashioned so that the entire rectangle lies within the boundaries of the page's printable area.

The *view* must be attached to a window; that is, it must be known to the Application Server. However, when printing, a BView can be asked to draw portions of its display that are not visible on-screen. Its drawing is not limited by the clipping region, its bounds rectangle, or the frame rectangles of ancestor views.

Any number of BViews can draw on a page if they are subjects of separate `DrawView()` calls.

`DrawView()` renders the entire hierarchy of BViews attached to the specified BView.

Note the following bugs (which will be fixed in subsequent releases):

- BeOS 5 (and earlier) renders only the first two hierarchy levels correctly.
- Hidden BViews (and views without `B_WILL_DRAW` specified) are rendered.

- BScrollBar and derived BViews aren't rendered.
- View background and overlay bitmaps are ignored; instead, the view color is printed.

After all views have drawn and the page is complete, **SpoolPage()** adds it to the spool file.**SpoolPage()** must be called once to terminate each page.

See also: **PrintableRect()**, **BView::Draw()**, "Drawing on the Page" in the class overview

---

FirstPage() , LastPage()

int32 **FirstPage**(void)
int32 **LastPage**(void)

These functions return the first and last pages that should be printed as part of the current job. If the pages are not set (for example, if the current job has been canceled), **FirstPage()** returns 0 and **LastPage()** returns a very large number (LONG_MAX).

GetResolution()

void **GetResolution**(int32 **xdpi*, int32 **ydpi*)

Fills the *xdpi* and *ydpi* variables with the X and Y resolution of the printer, in dots per inch.

LastPage() see FirstPage()

PaperRect() , PrintableRect()

BRect **PaperRect**(void)
BRect **PrintableRect**(void)

PaperRect() returns a rectangle that records the presumed size of the paper that the printer will use. Its left and top sides are at 0.0, so its right and bottom coordinates reflect the size of a sheet of paper. The size depends on choices made by the user when setting up the page layout.

PrintableRect() returns a rectangle that encloses the portion of a page where printing can appear. It's stated in the same coordinate system as the rectangle returned by **PaperRect()**, but excludes the margins around the edge of the paper. When drawing on the printed page, the left top corner of this rectangle is taken to be the coordinate origin, (0.0, 0.0).

The "Drawing Coordinates" section in the class overview illustrates these rectangles and their coordinate systems.

See also: **DrawView()**

PrinterType()

```
int32 PrinterType(void * = NULL)
```

Returns a code identifying whether the printer is color or black and white. The argument is currently not used.

The return value will be either **B_BW_PRINTER** or **B_COLOR_PRINTER**.

In the current release of BeOS, this function always returns **B_COLOR_PRINTER**, since there's no printer driver API for determining the printer type yet.

If an error occurs, this function will return an appropriate error code (for example, if the Print Server isn't running).

SetSettings() , Settings() , IsSettingsMessageValid()

```
void SetSettings(BMessage *configuration)
BMessage *Settings(void)
void IsSettingsMessageValid(BMessage *configuration) const
```

These functions set and return the group of parameters that define how a document should be printed. The parameters include some that capture the page layout of the document and some that define the current job. They're recorded in a BMessage object that can be regarded as a black box; the data in the message are interpreted by the Print Server and will be documented where the print driver API is documented.

Instead of looking in the **Settings()** BMessage, rely on BPrintJob functions to provide specific information about the layout and the print job. Currently, there are only two functions—**FirstPage()** and **LastPage()**, which return the first and last pages that need to be printed.

Settings() can be called to get the current configuration message, which can then be flattened and stored with the document. You can retrieve it later and pass it to **SetSettings()** to set initial configuration values the next time the document is printed, as discussed in the "Setting Up the Page Layout" and "Setting Up a Print Job" sections of the class overview.

If the message passed to settings doesn't contain all the information needed to properly configure the print job, the printer add-on being used will either determine appropriate default values or present a configuration dialog box.

SetSettings() assumes ownership of the object it's passed. If your application needs to retain the object, pass a copy to **SetSettings()**:

```
print_job_object.SetSettings(new BMessage(settings_message));
```

On the other hand, **Settings()** transfers ownership of the object it returns to the caller; you don't need to make a copy.

IsMessageSettingsValid() returns **true** if the specified message is a valid settings message; otherwise it returns **false**.

See also: **ConfigPage()**

SpoolPage() see DrawView()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPrintJob Function Summary

Function Summary

Function	Description
BeginJob()	Opens the spool file for the job
CanContinue()	Tests whether anything prevents the job from continuing
CancelJob()	Cancels print job and eliminates spool file
CommitJob()	Commits spooled pages to the printer
ConfigJob()	Asks Print Server to ask user to define the print job
ConfigPage()	Asks Print Server to ask user to configure the page
DrawView()	Display view on a page
FirstPage()	Returns number of first page to be printed
GetResolution()	Determines the printer's resolution
IsSettingsMessageValid()	Determines if a settings message is valid
LastPage()	Returns number of last page to be printed
PaperRect()	Returns paper size in a BRect
BPrintJob()	Constructor
~BPrintJob()	Destructor
PrintableRect()	Returns printable area on paper

	in a BRect
PrinterType()	Determines what type of printer will be used for the job
SetSettings()	Sets current printing configuration
Settings()	Returns current printing configuration
SpoolPage()	Add page to the spool file

The Be Book **The Interface Kit** **The Interface Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStatusBar

Derived from: public BView
Declared in: be/interface/StatusBar.h
Library: libbe.so
Summary: *more...*

A BStatusBar is a cosmetic object that graphically indicates the progress of an operation. It displays a horizontal bar that's filled with color from left to right as operation is progresses.

The label and text are abutted and aligned at the left of the bar; the trailing text and trailing label are abutted and aligned at the right of the bar.

The texts can change each time the bar is updated. The label and trailing label are set by the constructor and remain constant throughout the display, or until the object is reset for another operation.

To set the amount that the status bar is filled, you can call `Update()`, which lets you add a delta to the current value. The "filling" itself reflects the ratio (`current_value`)/(`max_value`), where `max_value` is 100.0 by default, but can be reset through `SetMaxValue()`. The minimum value of the bar is always 0.0.

You can call `Update()` directly, or invoke it indirectly (and asynchronously) by sending a `B_UPDATE_STATUS_BAR` message to the BStatusBar.

The `Reset()` function, which empties the bar can also be invoked through a message (`B_RESET_STATUS_BAR`).

Constructor and Destructor

BStatusBar()

```
BStatusBar(BRect frame, const char *name,
    const char *label = NULL, const char *trailingLabel = NULL)
BStatusBar(BMessage *archive)
```

Initializes the BStatusBar with a *label* and a *trailingLabel*, which can both be `NULL`. The *frame* rectangle is used for its origin and width only; when the BStatusBar is attached to a window, it will be resized to a height that precisely accommodates its parts. The *name* argument is passed on to the BView constructor.

The object is given a `B_WILL_DRAW` flag and a resizing mode that keeps it glued to the left and top sides of its parent.

The default font of the BStatusBar is the system plain font, and the default color of the bar is blue (50, 150, 255). It's initial value is 0.0, the minimum, and its default maximum value is 100.0.

See also: `SetMaxValue()`, `SetBarColor()`

~BStatusBar()

```
virtual ~BStatusBar()
```

Frees the labels and the text.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BStatusBar object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* doesn't contain data for an BStatusBar object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Archive()

```
virtual status_t Archive(BMessage *archive, bool deep = true) const
```

Calls the inherited version of `Archive()`, then adds the bar color, bar height, current value, and maximum value to the BMessage *archive*, along with the current text, trailing text, label, and trailing label.

AttachedToWindow()

```
virtual void AttachedToWindow(void)
```

Resizes the frame rectangle to accommodate the object's graphical parts. The width isn't altered.

This function also sets the view and low colors of the BStatusBar to match the background view color of its new parent.

BarColor() see SetBarColor()

BarHeight() see SetBarHeight()

CurrentValue() see SetMaxValue()

Draw()

```
virtual void Draw(BRect updateRect)
```

Draws the bar, labels, and text.

Label() , TrailingLabel()

```
const char *Label(void) const
const char *TrailingLabel(void) const
```

These functions return pointers to the object's label and trailing label. The returned strings belong to the BStatusBar object and should not be altered. They can be set only on construction or when all values are reset.

See also: `Reset()`

MaxValue() see SetMaxValue()

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Responds to **B_UPDATE_STATUS_BAR** and **B_RESET_STATUS_BAR** messages by calling the **Update()** and **Reset()** functions. Each message contains data that can be passed as arguments to the functions.

See also: **BView::MessageReceived()**, **Update()**, **Reset()**, "BStatusBar Messages" in the **Message Protocols** appendix

---

Reset()

virtual void **Reset(**const char **label* = NULL, const char **trailingLabel* = NULL**)**

Empties the status bar, sets its current value to 0.0 and its maximum value to 100.0, deletes and erases the text and trailing text, and replaces the label and trailing label with copies of the strings passed as arguments. If either argument is NULL, the label or trailing label will also be deleted and erased.

This gets the BStatusBar ready to be reused for another operation. For example, if several large files are being downloaded, the BStatusBar could be reset for each one.

You can call this function indirectly (and asynchronously) through a **B_RESET_STATUS_BAR** message. If the message has an entry named "label" containing a string (**B_STRING_TYPE**), the string will be passed to the function as the first argument. If there's a string in an entry named "trailing_label", it will be passed as the second argument. If either entry is absent, the value for the corresponding argument will be NULL.

See also: **SetText()**, **Update()**

SetBarColor() , BarColor()

virtual void **SetBarColor(**rgb_color *color***)**

rgb_color **BarColor(**void**)** const

These functions set and return the color that fills the bar to show how much of an operation has been completed. The default bar color is blue (50, 150, 255).

SetBarHeight() , BarHeight()

virtual void **SetBarHeight(**float *height***)**

float **BarHeight**(void) const

These functions set and return the height of the bar itself, minus the text and labels. The default height is 16.0 coordinate units. The frame rectangle is adjusted to accommodate the new height, and the object is redrawn.

SetMaxValue() , MaxValue() , CurrentValue()

virtual void **SetMaxValue**(float *max*)
float **MaxValue**(void) const
float **CurrentValue**(void) const

`SetMaxValue()` sets the maximum value of the BStatusBar, which by default is 100.0. `MaxValue()` returns the current maximum. The minimum value is 0.0 and cannot be changed.

`CurrentValue()` returns the current value of the BStatusBar. The current value is set by `Update()` and reset to 0.0 by `Reset()`.

The amount of "filling" in the status bar reflects the ratio `CurrentValue()`/`MaxValue()`.

See also: `Update()`, `Reset()`

SetText() , SetTrailingText() , Text() , TrailingText()

virtual void **SetText**(const char **string*)
virtual void **SetTrailingText**(const char **string*)
const char ***Text**(void) const
const char ***TrailingText**(void) const

These `SetText()` and `SetTrailingText()` functions erase the previous text or trailing text and replace them with copies of the *string* argument. *string* can be **NULL**. The view is automatically redrawn.

`Text()` and `TrailingText()` return pointers to the current text strings.

SetText())

TrailingLabel() see SetLabel()

TrailingText() see SetText()

Update()

virtual void **Update**(float *delta*, const char **text* = NULL, const char **trailingText* = NULL)

`Update()` updates the BStatusBar by adding *delta* to its current value and resetting its text and trailing text. Passing `NULL` for the *text* or *trailingText* argument retains the existing string(s). The status bar is automatically redrawn.

You can call `Update()` indirectly through a **`B_UPDATE_STATUS_BAR`** message that contains the following fields, which correspond to the function's arguments:

Field	Type
"delta"	B_FLOAT_TYPE
"text"	B_STRING_TYPE
"trailing text"	B_STRING_TYPE

To pass a **`NULL`** string through this message, simply don't add the "text" or "trailing text" field.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_high"	`B_FLOAT_TYPE`	Status bar height.
"_bcolor"	`B_INT32_TYPE`	Status bar color.
"_val"	`B_FLOAT_TYPE`	Current value.
"_max"	`B_FLOAT_TYPE`	Maximum value.
"_text"	`B_STRING_TYPE`	Status bar text.
"_ttext"	`B_STRING_TYPE`	Trailing text.
"_label"	`B_STRING_TYPE`	Status bar label.
"_tlabel"	`B_STRING_TYPE`	Trailing text label.

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if there is no trailing text, the "_ttext" field won't be found in the archive.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStatusBar Function Summary

Function Summary

Function	Description
Archive()	Archives a BStatusBar into a BMessage
AttachedToWindow()	Provided for implementation
BarColor()	Returns the completed fill color
BarHeight()	Returns the height of the bar
CurrentValue()	Returns the current value of the status bar
Draw()	Provided for implementation
Instantiate()	Creates a BStatusBar from a BMessage
Label()	Returns the object's label
MaxValue()	Returns the maximum value of the status bar
MessageReceived()	Provided for implementation
Reset()	Resets the status bar parameters
SetBarColor()	Sets the completed fill color
SetBarHeight()	Sets the height of the bar
SetMaxValue()	Sets the maximum value of the status bar
SetText()	Sets the status bar text
SetTrailingText()	Sets the status bar trailing text
BStatusBar()	Constructor
~BStatusBar()	Destructor

Text()	Returns the status bar text
TrailingLabel()	Returns the object's trailing label
TrailingText()	Returns the status bar trailing text
Update()	Updates the value of the status bar and redraws

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScreen

Derived from: *none*
Declared in: be/interface/Screen.h
Library: libbe.so
Allocation: Constructor or on the stack
Summary: *more...*

A BScreen object represents a single display screen that's connected to the computer. With a BScreen object you can...

- Get and set the screen's size and pixel depth.
- Get the screen's color map.
- Make a screen shot.
- Set the desktop color.
- Synchronize your code with the screen's retrace event.

You can't copy a BScreen object—the copy constructor and assignment operators are private.

Multiple Screens

Although, the BeOS currently only supports a single screen, in the future it will let the user hook up multiple screens. One of the screens, the *main screen*, will have the origin of the screen coordinate system at its left top corner. Other screens will be located elsewhere in the same coordinate system. If there's just one screen, it's the main screen.

A BScreen object represents one screen. An application can have more than one object referring to the same screen.

When multiple screens are supported, a **`screen_id`** identifier will be assigned to each one. Currently, **`B_MAIN_SCREEN_ID`** is the only identifier.

Constructor and Destructor

BScreen()

```
BScreen(BWindow *window)
BScreen(screen_id id = B_MAIN_SCREEN_ID)
```

Initializes the BScreen object so that it represents the screen where *window* is displayed or the screen identified by *id*. If *window* is **NULL** or hidden, or if the *id* is invalid, the BScreen will represent the main screen.

Since multiple monitors aren't currently supported, there's no API for screen identifiers other than for the main screen.

To be sure the new object was correctly constructed, call **IsValid()**.

~BScreen()

```
~BScreen()
```

Unlocks the screen and invalidates the BScreen object.

Member Functions

ColorForIndex() see ColorMap()

ColorMap() , IndexForColor() , ColorForIndex()

const **color_map** ***ColorMap(**void**)**
inline uint8 **IndexForColor(rgb_color** *color***)** uint8 **IndexForColor(**uint8 *red*, uint8 *green*, uint8 *blue*, uint8 *alpha* = 255**)**
rgb_color ColorForIndex(const uint8 *index***)**
uint8 **InvertIndex(**uint8 *index***)**

These functions return information from the **color_map** structure for this screen. The **color_map** structure defines the set of 256 colors that can be displayed in an **B_CMAP8** color space. A single **color_map** is shared by all applications that display on the same screen. See the **color_map** structure for more information about the structure.

ColorMap() returns a pointer to the **color_map** itself. The structure belongs to the BScreen object; you can't modify or free it. (Note that the the **system_colors()** function retrieves the **color_map** structure for the main screen without reference to a BScreen object.)

IndexForColor() returns the "index" of the 8-bit color that, in this screen's color map, most closely matches the given 32-bit color. You can pass the index to functions such **BBitmap::SetBits()** to set an 8-bit color. Note that **IndexForColor()** knows how to convert **B_TRANSPARENT_32_BIT** into **B_TRANSPARENT_8_BIT**.

ColorForIndex() returns the 32-bit color representation of a given 8-bit color *index*. This function doesn't convert **B_TRANSPARENT_8_BIT** into **B_TRANSPARENT_32_BIT**.

InvertIndex() takes an 8-bit *index* and returns an index that represents the color's "inversion." Inverted colors are typically used for highlighting.

The information gained through **IndexForColor()**, **ColorForIndex()**, and **InvertIndex()** can be retrieved more efficiently from the **color_map** structure. If you're repeatedly calling these functions, you should consider accessing the **color_map** structure, instead. Note, however, that the intelligent **B_TRANSPARENT_32_BIT** to **B_TRANSPARENT_8_BIT** conversion is not supported by the structure.

ColorSpace()

color_space ColorSpace(void**)**

Returns the color space of the screen display—typically **B_CMAP8**, **B_RGB15**, or **B_RGB32**—or **B_NO_COLOR_SPACE** if the BScreen object is invalid.

The color space is set by the user through the **Screen** preferences application. You can set it programatically through the **SetMode()** function.

- DesktopColor() see SetDesktopColor()
- DPMSCapabilities() see SetDPMS()
- DPMSState() see SetDPMS()

- Frame()

```
BRect Frame(void)
```

Returns the rectangle that locates the screen in the screen coordinate system. For example, the frame for a 1,024 * 768 main screen looks like this:

```
BRect(0.0, 0.0, 1023.0, 767.0)
```

If the BScreen object is invalid, all sides of the rectangle are set to 0.0.

The screen's frame rectangle is set by the user through the **Screen** preferences application. You can set it programatically through the **SetMode()** function.

- GetBitmap() see ReadBitmap()

- GetDeviceInfo

```
status_t GetDeviceInfo(accelerant_device_info *info)
```

Returns information about the graphics card.

- GetMode() see GetModeList()

- GetModeList() , SetMode() , GetMode()

```
status_t GetModeList(display_mode **mode_list, uint32 *count)
status_t SetMode(display_mode *mode, bool makeDefault = false)
status_t GetMode(display_mode *mode)
```

These functions set and get the screen's *display mode*. Each **display_mode** structure (defined in **add-ons/graphics/Accelerant.h**) is a distinct combination of screen size, pixel depth, and display timing.

GetModeList() allocates and returns, in *mode_list*, a list of the **display_mode** structures that the graphics card is guaranteed to support; *count* is set to the number of **display_mode** elements in the list. The caller is responsible for freeing *mode_list*.

There's no guarantee that the monitor can support all of the modes that GetModeList() retrieves.

SetMode() resets the screen to the given *mode*. If *makeDefault* is true, the mode becomes the default for the current workspace.

GetMode() copies the current **display_mode** into *mode*.

The display_mode structure is:

```
typedef struct {
    display_timing timing;
    uint32 space;
    uint16 virtual_width;
    uint16 virtual_height;
    uint16 h_display_start;
    uint16 v_display_start;
    uint32 flags;
    } display_mode;
```

- **timing** provides CTRC timing information.
- **space** is the color space of the display.
- **virtual_width** is the screen's virtual width in pixels, and **virtual_height** is the screen's virtual height in lines.
- **h_display_start** is the first displayed pixel in a line, and **v_display_start** is the first displayeed line.
- **flags** are mode flags:
-

Constant	Meaning
B_SCROLL	Scrolling display; a large display is being simulated by scrolling around on a smaller screen.
B_8_BIT_DAC	The DAC is in 8-bit mode.
B_HARDWARE_CURSOR	The mode supports a hardware cursor.
B_PARALLEL_ACCESS	The mode supports parallel access.
B_DPMS	The mode supports power management.

B_IO_FB_NA	The graphics card's frame buffer shouldn't be touched by the Application Server while the card's acceleration engine might be doing so.

- The display_timing structure is:

```
typedef struct {
    uint32 pixel_clock;
    uint16 h_display;
    uint16 h_sync_start;
    uint16 h_sync_end;
    uint16 h_total;
    uint16 v_display;
    uint16 h_display;
    uint16 v_sync_start;
    uint16 v_sync_end;
    uint16 v_total;
    uint32 flags;
    } display_timing;
```

- **pixel_clock** is in kHz.
- **h_display** is in pixels, not in character clocks.
- **v_display** is in lines.
- **flags** are:
-

Constant	Meaning
B_BLANK_PEDESTAL	Use a 7.5 IRE blanking pedestal instead of a 0.0 IRE blanking pedestal. Usually 0.0 IRE.
B_TIMING_INTERLACED	The mode is interlaced instead of progressively scanned. Rarely set; most modern displays don't need this.
B_POSITIVE_HSYNC	If set, the mode uses a positive (high) sync polarity.
B_POSITIVE_VSYNC	If set, the mode uses a negative (low) sync polarity.
B_SYNC_ON_GREEN	The mode generates sync information on the green color signal.

See also: ProposeMode()

GetPixelClockLimits()

```
status_t GetPixelClockLimits(display_mode *mode,
    uint32 *low,
    uint32 *high)
```

This function returns, in *low* and *high*, the minimum and maximum "pixel clock" rates (in thousands-of-pixels per second) that are possible for the given mode. Given the pixel clock and a display mode, you can determine the refresh rate range by dividing the pixel clock by the "real" size of the screen, thus:

```
uint32 hi_clock, lo_clock;
float hi_refresh, lo_refresh;
float real_size;
display_mode mode;
GetMode(&mode);
GetPixelClockLimits(&mode, &lo_clock, &hi_clock);

/* The real screen dimensions (i.e. the dimensions for the purposes
* of the gun) are given by the 'timing.h_total' and
* 'timing.v_total' fields.
*/
total_size = mode.timing.h_total * mode.timing.v_total

/* Get the refresh rate by dividing the pixel clock by the total
* screen size. Remember -- the pixel clock values are given in
* kHz; we multiply by 1000.0 to retrieve refresh rates in Hz.
*/
hi_refresh = ((float)hi_clock*1000.0)/(float)total_size;
lo_refresh = ((float)lo_clock*1000.0)/(float)total_size;
```

RETURN CODES

- **B_OK**. Pixel clock limits returned successfully.
- **B_ERROR**. No clock limits known.

GetTimingConstraints()

```
status_t GetTimingConstraints(display_timing_constraints *dtc)
```

This function fills out the *dtc* structure with the timing constraints of the current display mode.

RETURN CODES

- **B_OK**. Constraints returned successfully.
- **B_ERROR**. No constraints known.

ID()

```
screen_id ID(void)
```

Returns the identifier for the screen. The main screen is identified as **B_MAIN_SCREEN_ID**.

The ID isn't presistent across boots, and may change if the monitor is diconnected and then reconnected.

Currently, this function always returns **B_MAIN_SCREEN_ID**, even if the BScreen object is invalid.

IndexForColor() see ColorMap()

InvertIndex() see ColorMap()

IsValid()

```
bool IsValid(void)
```

Returns **true** if the BScreen object is valid (if it represents a real screen connected to the computer), and **false** if not.

ProposeMode()

```
status_t ProposeMode(display_mode *candidate,
    const display_mode *low,
    const display_mode *high)
```

ProposeMode() is a convenience function that attempts to adjust *candidate* so that it's a supported mode (as listed by the **GetModeList()** function). It then compares the possibly-adjusted *candidate* to the limits declared in *low* and *high* and expresses this comparison in the return value. Note that the function *doesn't* adjust *candidate* so that it is, of necessity, between *low* and *high*.

Exactly how **ProposeMode()** works is up to the individual graphics driver. It's expected that the function will adjust *candidate*'s screen size fields while holding the color space constant.

This function was formerly called `ProposeDisplayMode()`.

RETURN CODES

- **B_OK**. *candidate* (as returned) is supported and falls within the limits.
- **B_BAD_VALUE**. *candidate* (as returned) is supported, but doesn't fall within the limits.
- **B_ERROR**. *candidate* isn't supported.

See also: GetModeList()

ReadBitmap() , GetBitmap()

```
status_t ReadBitmap(BBitmap *buffer,
    bool draw_cursor = true,
    BRect *bounds = NULL)
status_t GetBitmap(BBitmap **buffer,
    bool draw_cursor = true,
    BRect *bounds = NULL)
```

These functions provide read-only access to the screen by copying the screen's contents into the first argument BBitmap. The difference between them is that **ReadBitmap()** expects you to allocate the BBitmap before passing it in, while **GetBitmap()** allocates a new BBitmap for you. The caller is responsible for freeing the BBitmap allocated by **GetBitmap()**.

The *draw_cursor* argument determines whether the cursor is drawn in the screen shot; *bounds* let you specify the region, in screen coordinates, that you want copied. If *bounds* is **NULL**, the entire screen is copied. The functions fail if the **bounds** rectangle doesn't fall wholly within the screen's frame.

The functions return **B_OK** on success or **B_ERROR** on failure.

SetDesktopColor() , DesktopColor()

```
void SetDesktopColor(rgb_color color, bool makeDefault = true)
rgb_color DesktopColor(void)
```

These functions set and return the color of the desktop—the backdrop against which windows are displayed on the screen. **SetDesktopColor()** makes an immediate change in the desktop color displayed on-screen; **DesktopColor()** returns the color currently displayed.

If the *makeDefault* flag is **true**, the color that's set becomes the default color for the screen; it's the color that will be shown the next time the machine is booted. If the flag is **false**, the color is set only for the current session.

The "Background Images" section tells you how to convince the desktop to display a bitmap image.

Typically, users choose the desktop color with the **Screen** preferences application.

SetDPMS() , DPMSState() , DPMSCapabilities()

```
status_t SetDPMS(uint32 dpmsState)
uint32 DPMSState(void)
uint32 DPMSCapabilities(void)
```

SetDPMS() lets you set the VESA Display Power Management Signaling state for the screen. The state can be one of the following values:

Constant	Meaning
B_DPMS_ON	Image is visible, normal screen operation.
B_DPMS_STAND_BY	Image is not visible, but can be restored "instantly." Saves around 30% of the power used by the monitor in B_DPMS_ON mode.
B_DPMS_SUSPEND	Image is not visible, but can be restored in less than five seconds. Saves more power by turning off the CRT's heater. The amount of savings (or if there's any) depends on the display.
B_DPMS_OFF	Image is not visible and will take some time to restore. Typically turns off all monitor power except the processor watching the sync signals for a higher power state (typically B_DPMS_ON).

DPMSState() returns the current display state, indicating whether the monitor is on or off or in one of the two sleep modes.

DPMSCapabilities() indicates which of the above modes the monitor supports.

SetMode() see GetModeList()

SetToNext

```
status_t SetToNext(void)
```

In the current BeOS release, this function always returns B_ERROR.

WaitForRetrace()

```
status_t WaitForRetrace(void)
status_t WaitForRetrace(bigtime_t timeout)
```

Blocks until the monitor has finished the current vertical retrace, then returns B_OK. There are a few milliseconds available before it begins another retrace. Drawing changes made to the frame buffer in this period won't cause any "flicker" on-screen.

For some graphics card drivers, this function will wait for vertical sync; for others it will wait until vertical blank, providing a few extra milliseconds.

The *timeout* argument lets you provide a timeout in microseconds—if the screen hasn't retraced within the limit, the function returns B_ERROR.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScreen Function Summary

Function Summary

Function	Description
ColorForIndex()	Returns an index to the color map matching a given color
ColorMap()	Returns a pointer to the color map for the screen
ColorSpace()	Returns the color space of the screen
DPMSCapabilities()	Returns the screen's DPMS capabilities
DPMSState()	Returns the current DPMS state
DesktopColor()	Returns the desktop background color
Frame()	Returns a rectangle giving the screen dimensions
GetBitmap()	Returns a new BBitmap initialized with the screen contents
ID()	Returns the identifier for the screen
IndexForColor()	Returns an index to the color map matching a given color
IsValid()	Tests whether the object initialized properly
ReadBitmap()	Copies the screen contents into a BBitmap
BScreen()	Constructor
~BScreen()	Destructor
SetDPMS()	Sets the DPMS state
SetDesktopColor()	Sets the desktop background color

WaitForRetrace()	Blocks until the monitor has finished the current vertical retrace

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Background Images

Declared in: be/be_apps/Tracker/Background.h

Tracker lets you put background images on each workspace on your desktop and in the background of individual Tracker folder windows.

Currently, Tracker only paints the background of a folder window when the window is in icon (or mini-icon) view mode.

Background Attributes

Information about the background image for a folder is stored as a flattened BMessage in the folder's **B_BACKGROUND_INFO** file attribute. The **B_DESKTOP_FOLDER** directory holds the background info for the workspaces (one flattened message per workspace).

The **B_BACKGROUND_INFO** attribute's data is a flattened BMessage with these fields:

Name	Type	Description
B_BACKGROUND_IMAGE	B_STRING_TYPE	The full path to the image. The image needs to be in a format understood by one of the translators (see the Translation Kit) installed on the system.
B_BACKGROUND_MODE	B_INT32_TYPE	Specifies how the background is placed (tiled, centered, etc). See the list of mode constants after this table.
B_BACKGROUND_ORIGIN	B_POINT_TYPE	If **B_BACKGROUND_MODE** is **B_BACKGROUND_MODE_USE_ORIGIN**, the image will be drawn with the upper left corner at this point (which can be anything, even negative).
B_BACKGROUND_ERASE_TEXT	B_BOOL_TYPE	If **true** the icon label will have its background erased to the window's background color, otherwise the label will be drawn on top of the image (which could make it difficult to read).
B_BACKGROUND_WORKSPACES	B_UINT32_TYPE	A bitmask indicating which workspaces the image should be drawn on. This only applies to Desktop images.

The **B_BACKGROUND_MODE** values are:

- **B_BACKGROUND_MODE_USE_ORIGIN**. Draw the image once at the location specified by **B_BACKGROUND_ORIGIN**, relative to the upper-left corner of the window or Desktop.
- **B_BACKGROUND_MODE_CENTERED** (only works on Desktop) Center the image.
- **B_BACKGROUND_MODE_SCALED** (only works on Desktop) Scale the image to fit.
- **B_BACKGROUND_MODE_TILED** Tile the image to cover the entire window/Desktop.

You can write multiple sets of these message fields into the Desktop's **B_BACKGROUND_INFO** attribute message in order to put different images on different workspaces. The fields are correlated by index. Don't put different **B_BACKGROUND_IMAGE** paths onto the same workspace though, the behavior is undefined.

For example, this is OK:

```
...
/* ALMOST_ALL_WORKSPACES is every workspace except the first */
const int32 ALMOST_ALL_WORKSPACES = 0xffffffff - 0x00000001;
```

```
const int32 OTHER_WORKSPACE = 0x00000001;
...
BMessage backgrounds;

/* Add the first image to ALMOST_ALL_WORKSPACES */
backgrounds.AddString( B_BACKGROUND_IMAGE, "/boot/home/nice.jpg" );
backgrounds.AddInt32( B_BACKGROUND_WORKSPACES, ALMOST_ALL_WORKSPACES );
backgrounds.AddInt32( B_BACKGROUND_MODE, B_BACKGROUND_MODE_TILED );
backgrounds.AddPoint( B_BACKGROUND_ORIGIN, BPoint( 0.0, 0.0 ) );
backgrounds.AddBool( B_BACKGROUND_ERASE_TEXT, true );

/* Add the second image to the OTHER_WORKSPACE */
backgrounds.AddString( B_BACKGROUND_IMAGE, "/boot/home/nicer.jpg" );
backgrounds.AddInt32( B_BACKGROUND_WORKSPACES, OTHER_WORKSPACE );
backgrounds.AddInt32( B_BACKGROUND_MODE, B_BACKGROUND_MODE_SCALED );
backgrounds.AddPoint( B_BACKGROUND_ORIGIN, BPoint( 0.0, 0.0 ) );
backgrounds.AddBool( B_BACKGROUND_ERASE_TEXT, true );
...
```

This is *not* OK, and results in undefined behaviour:

```
...
const int32 ALL_WORKSPACES = 0xffffffff;
...
BMessage backgrounds;

backgrounds.AddString( B_BACKGROUND_IMAGE, "/boot/home/nice.jpg" );
backgrounds.AddInt32( B_BACKGROUND_WORKSPACES, ALL_WORKSPACES );
/* ... other B_BACKGROUND_* data here ... */

backgrounds.AddString( B_BACKGROUND_IMAGE, "/boot/home/nicer.jpg" );
backgrounds.AddInt32( B_BACKGROUND_WORKSPACES, ALL_WORKSPACES );
/* ... other B_BACKGROUND_* data here ... */
...
```

The **B_BACKGROUND_INFO** attribute's contents might be extended in the future (but will remain compatible with the existing attribute data). Be prepared to preserve any data you don't recognize by reading in the existing **B_BACKGROUND_INFO** attribute (if one exists).

Special Folders

There are two special folders you can apply attributes to:

Directory	Description
B_DESKTOP_DIRECTORY	Applying the **B_BACKGROUND_INFO** attribute to **B_DESKTOP_DIRECTORY** (see find_directory() in the Storage Kit) will change the Desktop's background image (but see below to find out how to tell the Tracker to update the image).
Tracker/DefaultFolderTemplate inside **B_USER_SETTINGS_DIRECTORY**	Applying the **B_BACKGROUND_INFO** attribute to this directory sets the default folder background. Any folder that doesn't have its own **B_BACKGROUND_INFO** will display this image.

Notifying Tracker

Once you've written an appropriate **B_BACKGROUND_INFO** attribute, you'll want to tell the Tracker to pick up the new background image.

Send the Tracker's main looper (remember, the Tracker's signature is "application/x-vnd.Be-TRAK") a BMessage with a *what* of **B_RESTORE_BACKGROUND_IMAGE** to tell it to reload the desktop background image.

Sending the same message to the looper for a specific Tracker window will tell *that window* to reload its background image. This is a little trickier because you need to find the right window.

Refreshing All Windows

Taking the shotgun approach, you can tell all of the open Tracker windows to repaint their backgrounds by using Tracker scripting (see Scripting in the Application Kit). You'll need to do this if you change the default folder background image.

This code asks the Tracker for windows that have a "Poses" property and sends them a **B_RESTORE_BACKGROUND_IMAGE**. This updates every Tracker window that's displaying a folder's contents.

```
BMessenger tracker( "application/x-vnd.Be-TRAK" );

int32 i = 0;
BMessage reply;
int32 err;

do {
   /* scripting message */
   BMessage msg( B_GET_PROPERTY );

   /* look at the "Poses" in every Tracker window */
   msg.AddSpecifier( "Poses" );
   msg.AddSpecifier( "Window", i++ );

   reply.MakeEmpty();

   tracker.SendMessage( &msg, &reply );

   /* break out of the loop when we're at the end of
   * the windows
   */
   if( reply.what == B_MESSAGE_NOT_UNDERSTOOD &&
      reply.FindInt32( "error", &err ) == B_OK &&
      err == B_BAD_INDEX )
      break;
```

```
   /* don't stop for windows that don't understand
   * a request for "Poses"; they're not displaying
   * folders
   */
   if( reply.what == B_MESSAGE_NOT_UNDERSTOOD &&
      reply.FindInt32( "error", &err ) == B_OK &&
      err != B_BAD_SCRIPT_SYNTAX )
      continue;

   BMessenger m;
   if( reply.FindMessenger( "result", &m ) == B_OK) {
      /* m is the messenger of a Tracker window with
         * a folder inside
         */

      m.SendMessage( B_RESTORE_BACKGROUND_IMAGE );
   }

} while( true );
```

Refreshing One Window

If want to refresh the background of the folder you've changed, you scan through the open folders using code similar to the above, until you find the folder with the "Path" you're interested in. Once you've got the right window, you send it a **B_RESTORE_BACKGROUND_IMAGE** message. Using this technique is nicer (from the user's point of view) because only one Tracker window gets updated.

Here's the same code, modified to target a specific path. Assume you've still got the path in question in *the_path* (you just used it as a **B_BACKGROUND_IMAGE**):

```
BMessenger tracker( "application/x-vnd.Be-TRAK" );

/* CHANGE: we'll increment "i" at the start of the loop
* now; this lets us use it later when we find
* a Window with a Poses
*/
int32 i = -1;
BMessage reply;
int32 err;

do {
   /* CHANGE: */
   i++;

   /* scripting message */
   BMessage msg( B_GET_PROPERTY );
```

```
/* look at the "Poses" in every Tracker window */
msg.AddSpecifier( "Poses" );
msg.AddSpecifier( "Window", i ); /* CHANGE */

reply.MakeEmpty();

tracker.SendMessage( &msg, &reply );

/* break out of the loop when we're at the end of
* the windows
*/
if( reply.what == B_MESSAGE_NOT_UNDERSTOOD &&
   reply.FindInt32( "error", &err ) == B_OK &&
   err == B_BAD_INDEX )
   break;

/* don't stop for windows that don't understand
* a request for "Poses"; they're not displaying
* folders
*/
if( reply.what == B_MESSAGE_NOT_UNDERSTOOD &&
   reply.FindInt32( "error", &err ) == B_OK &&
   err != B_BAD_SCRIPT_SYNTAX )
   continue;

/* CHANGE: everything from here down is new */

/* make a messenger to this window */
BMessenger m;
if( reply.FindMessenger( "result", &m ) != B_OK) continue;

/* found a Window with a Poses, ask for its Path */
msg.MakeEmpty();
msg.what = B_GET_PROPERTY;
msg.AddSpecifier( "Path" );
msg.AddSpecifier( "Poses" );
msg.AddSpecifier( "Window", i );

reply.MakeEmpty();

tracker.SendMessage( &msg, &reply );

/* go on with the next if this din't have a Path */
if( reply.what == B_MESSAGE_NOT_UNDERSTOOD ) continue;

/* dig out the Path */
entry_ref ref;
```

```
   if( reply.FindRef( "result", &ref ) == B_OK ) {
      BEntry ent( &ref );
      BPath path( &ent );

      /* these are not the Paths you're looking for */
      if( strcmp( the_path, path.Path() ) ) continue;
   }

   /* if you got this far, you have a messenger to a valid Tracker
   * window displaying a folder with the path you were after,
   * so tell it to refresh its background
   */
   m.SendMessage( B_RESTORE_BACKGROUND_IMAGE );

} while( true );
```

The Be Book

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTab

Derived from: public BArchivable
Declared in: be/interface/TabView.h
Library: libbe.so
Summary: *more...*

The BTab class defines the tabs used by the BTabView class. Each tab is represented by a single BTab object, which is called to render and manage the tab.

When a tab is created, a target view is specified as a parameter to the BTab constructor, or by calling `SetView()`. The target view is the view that will be displayed in the BTabView's container view when the BTab is selected.

Users select tabs by clicking on them, or by using keyboard navigation to focus and select the tab.

An example of how to create a BTabView and attach BTab objects to it is given in the BTabView section.

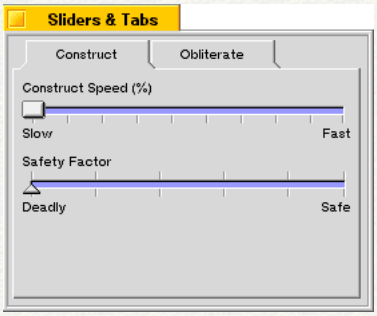
Customizing the Appearance of a BTab

Customizing the appearance of your tabs is achieved by overriding the `DrawTab()`, DrawFocusMark(), and/or `DrawLabel()` functions.

These functions are responsible for all drawing of the BTab. `DrawTab()` renders the entire tab, excluding the focus mark: it draws the borders and calls `DrawLabel()` to render the text of the label.

`DrawFocusMark()` draws the indicator that shows which tab is the current focus for keyboard navigation.

By default, tabs have a beveled, rounded look. Let's look at an example in which we replace this appearance with a square shape:



To do this, we create a new class, derived from BTab, that overrides the `DrawTab()` function.

```
class CustomTab : public BTab {
   public:
      virtual void      DrawTab(BView *owner, BRect frame,
                        tab_position position, bool full=true);
};
```

The `DrawTab()` function is implemented as follows:

```
const rgb_color kWhite = {255,255,255,255};
const rgb_color kGray = {219,219,219,255};
const rgb_color kMedGray = {180,180,180,255};
const rgb_color kDarkGray = {100,100,100,255};
const rgb_color kBlackColor = {0,0,0,255};

void CustomTab::DrawTab(BView *owner, BRect frame, tab_position position,
                bool full) {
   rgb_color      hi;
   rgb_color      lo;

   // Save the original colors
```

```
hi = owner->HighColor();
lo = owner->LowColor();

// Draw the label by calling DrawLabel()

owner->SetHighColor(kBlackColor);
owner->SetLowColor(kGray);
DrawLabel(owner, frame);

// Start a line array to draw the tab --
// this is faster than drawing the lines
// one by one.

owner->BeginLineArray(7);

// Do the bottom left corner, visible
// only on the frontmost tab.

if (position != B_TAB_ANY) {
   owner->AddLine(BPoint(frame.left, frame.bottom),
            BPoint(frame.left+3, frame.bottom-3), kWhite);
}

// Left wall -- always drawn

owner->AddLine(BPoint(frame.left+4, frame.bottom-4),
         BPoint(frame.left+4, frame.top), kWhite);

// Top -- always drawn

owner->AddLine(BPoint(frame.left+5, frame.top),
         BPoint(frame.right-5, frame.top), kWhite);

// Right wall -- always drawn. Has a nice bevel.

owner->AddLine(BPoint(frame.right-4, frame.top),
         BPoint(frame.right-4, frame.bottom-4), kDarkGray);
owner->AddLine(BPoint(frame.right-5, frame.top),
         BPoint(frame.right-5, frame.bottom-4), kMedGray);

// Bottom-right corner, only visible if the tab
// is either frontmost or the rightmost tab.

if (full) {
   owner->AddLine(BPoint(frame.right-3, frame.bottom-3),
            BPoint(frame.right, frame.bottom), kDarkGray);
   owner->AddLine(BPoint(frame.right-4, frame.bottom-3),
            BPoint(frame.right-1, frame.bottom), kMedGray);
```

```
    }

    owner->EndLineArray();

    owner->SetHighColor(hi);
    owner->SetLowColor(lo);
}
```

The *owner* parameter is a pointer to the BView in which the tab is drawn. *frame* is the frame rectangle of the tab; the tab should be drawn to fill this rectangle. The *position* parameter, which can be one of the following values, specifies the placement of the tab, to assist in making intelligent decisions on which parts of the tab are visible and which are not:

B_TAB_FIRST	The tab is the leftmost tab (the one with index 0 in the BTabView).
B_TAB_FRONT	The tab is the frontmost tab.
B_TAB_ANY	The tab neither the frontmost nor the leftmost tab.

This is a fairly trivial example, and is self-explanatory—with two caveats:

```
    // Do the bottom left corner, visible
    // only on the frontmost tab.

    if (position != B_TAB_ANY) {
        owner->AddLine(BPoint(frame.left, frame.bottom),
                BPoint(frame.left+3, frame.bottom-3), kWhite);
    }
```

This code is responsible for drawing the portion of the tab that connects to the box surrounding the BTabView's container. In our custom BTab, this is simply a diagonal line that extends from the bottom left corner of the frame upward and inward slightly.

However, this portion of the tab is only visible on the first or frontmost tab, so we only draw this segment if its *position* isn't **B_TAB_ANY** (in other words, if its position is either **B_TAB_FRONT** or **B_TAB_FIRST**).

```
    // Bottom-right corner, only visible if the tab
    // is either frontmost or the rightmost tab.

    if (full) {
        owner->AddLine(BPoint(frame.right-3, frame.bottom-3),
                BPoint(frame.right, frame.bottom), kDarkGray);
        owner->AddLine(BPoint(frame.right-4, frame.bottom-3),
                BPoint(frame.right-1, frame.bottom), kMedGray);
    }
```

This code, which draws the lower-right corner of the tab (where it meets back up with the box surrounding the container view), only runs if the *full* parameter is **true**. This is because the

right edge of a tab can be obscured by the tab to its left.

Hook Functions

Deselect()

Can be augmented if your derived class needs to perform additional functions when a tab is deselected. You should always call the inherited version of the function once your additional code has run.

DrawFocusMark()

Can be overridden to draw the mark indicating that the tab is the current focus. This lets you customize the appearance of your tabs.

DrawTab()

Can be overridden to customize the appearance of your tabs. This function should call **DrawLabel()** and draw the tab itself.

DrawLabel()

Can be overridden to customize the appearance of your tabs. This should simply draw the string returned by the **Label()** function as the tab's title.

Select()

Can be augmented if your derived class needs to perform additional tasks when a tab is selected. You should always call the inherited version of the function before your implementation returns.

SetEnabled()

Can be augmented if your derived class needs to perform additional actions when a tab is enabled or disabled. You should always call the inherited version of the function before your implementation returns.

SetLabel()

Can be augmented to take notice when the tab's label is changed. You should always call the inherited version of the function before your implementation returns.

SetView()

Can be augmented to take notice when the target view of a tab is changed. You should always call the inherited version of the function before your implementation returns.

Constructor and Destructor

BTab()

```
BTab(BView * tabView = NULL)
BTab(BMessage *archive)
```

Initializes the BTab to be enabled, but neither selected nor the current focus. The specified view becomes the tab's target view—when the tab is selected, its target view is activated. See the BTabView class for further details on how this works.

If an *archive* message is specified, the message's contents are used to duplicate the archived BTab object.

See also: `SetView()`

~BTab()

```
virtual ~BTab()
```

Frees all memory the BTab allocated. If there is a target view assigned to the tab, it is removed from the parent window and deleted.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BTab object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the message doesn't contain archived data for a BTab, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **`Archive()`** and stores the BTab in the BMessage *archive*.

See also: **`BArchivable::Archive()`**, **`Instantiate()`** static function

DrawFocusMark() , DrawTab() , DrawLabel()

virtual void **DrawFocusMark(**BView **owner*, BRect *frame***)**
virtual void **DrawTab(**BView **owner*, BRect *frame*, tab_position *position*, bool *full* = **true)**
virtual void **DrawLabel(**BView **owner*, BRect *frame***)**

These three functions can be implemented by your BTab-derived class to create a new visual appearance for your application's tabs. The *owner* is the BView in which your tab is being drawn, and the *frame* is the rectangle in which the tab is to be drawn.

The `DrawFocusMark()` function draws the mark indicating that the BTab object is in focus. By default, this consists of a line in the keyboard navigation color, drawn across the bottom of the tab's frame rectangle.

`DrawTab()` is called to draw the tab. It draws the tab's title by calling `DrawLabel()`, then renders the lines to create the tab itself. The `position` of the tab may affect how the tab is rendered—for example, if the tab is frontmost, it may have a different appearance than the other tabs.

If *full* is true, the complete tab is drawn inside the *frame* rectangle. If full is false, the right side of the tab is being obscured by the tab to its left, so the right edge should be eliminated or truncated as necessary.

IsEnabled() , SetEnabled()

bool **IsEnabled(**void**)** const
virtual void **SetEnabled(**bool *enabled***)**

The `IsEnabled()` function returns the **`true`** if the tab is enabled (and can therefore be selected by the user) or **`false`** if the tab is disabled.

`SetEnabled()` is called to enable or disable the tab. Pass a value of **`true`** to enable the tab, or **`false`** to disable it.

IsSelected() , Deselect() , Select()

bool **IsSelected(**void**)** const
virtual void **Deselect(**void**)**
virtual void **Select(**BView **owner***)**

The `IsSelected()` function returns **`true`** if the tab is currently selected, **`false`** if it's not.

`Deselect()` is called to deselect the tab. This removes the tab's target view from the owner window by calling the target view's **`RemoveSelf()`** function.

`Select()` is called to select the tab. This also adds the tab's target view to the specified owner view. This is called after the previously selected tab's `Deselect()` function is called.

Label() , SetLabel()

const char ***Label(**void**)** const
virtual void **SetLabel(**const char **label***)**

The `Label()` function returns the tab's label. The label is the same as the target view's name.

`SetLabel()` is called to set the tab's label. This also changes the target view's name to match the tab's label, if a target view exists.

If the tab doesn't have a target view, `SetLabel()` does nothing. Make sure a target view has been set (by calling **`SetView()`**, **`BTabView::AddTab()`** with a valid target view argument, or in the BTab constructor) before you call `SetLabel()`.

IsFocus() , MakeFocus()

bool **IsFocus(**void**)** const
void **MakeFocus(**bool *inFocus* = **`true`****)**

`IsFocus()` returns **`true`** if the tab is the current focus or **`false`** if it is not.

`MakeFocus()` specifies whether or not the tab is the current focus. Pass **`true`** to make the tab the current focus, or **`false`** if you don't want it to be the focus.

View() , SetView()

BView ***View(**void**)** const
virtual void **SetView(**BView **view***)**

The `View()` function returns the tab's target view.

`SetView()` is called to set the tab's target view to the specified *view*.

Archived Fields

The `Archive()` function adds the following field to its BMessage argument:

Field	Type code	Meaning
"_disable"	`B_BOOL_TYPE`	**true** if the tab is disabled. Not present if the tab is enabled.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTab Function Summary

Function Summary

Function	Description
Archive()	Archives the BTab into a BMessage
Deselect()	Deselects the tab
DrawFocusMark()	Draws the focus mark
DrawLabel()	Draws the tab label
DrawTab()	Draws the tab
Instantiate()	Creates a BTab from a BMessage
IsEnabled()	Tests whether the tab is enabled
IsFocus()	Tests whether the tab is the current focus
IsSelected()	Tests whether the tab is currently selected
Label()	Returns the tab's label
MakeFocus()	Sets whether the tab is the current focus
Select()	Selects the tab
SetEnabled()	Enables or disables the tab
SetLabel()	Sets the tab's label
SetView()	Sets the tab's target view
BTab()	Constructor
~BTab()	Destructor
View()	Returns the tab's target view

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

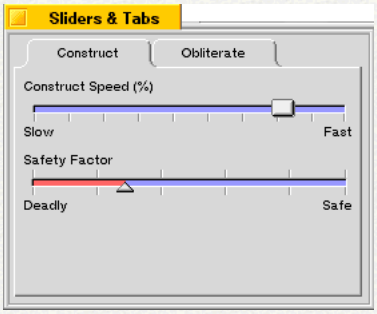
BTabView

Derived from: public BView
Declared in: be/interface/TabView.h
Library: libbe.so
Summary: *more...*

The tab view is a fairly common construct in modern user interfaces. The BTabView class serves as a container for a series of BTab objects as well as their target views; the tabs run horizontally across the top of the view from left to right, and beneath them is a container view in which the currently-selected tab's target view is presented. A box is drawn around the container view to clearly define the relationship between the tabs and the container view.

Creating a BTabView

In the figure below, we see a window that contains a BTabView with two BTab objects attached, called "Construct" and "Obliterate." The "Construct" tab is currently selected, and its target view is visible. The target view for the "Construct" tab contains two BSlider objects (these sliders are used as an example in the BSlider section and you can learn more about them there).



The BTabView in this window is constructed using the following code, which can be used in the constructor of the parent window:

```
BRect r;
BTabView *tabView;
BTab *tab;

r = Bounds();
r.InsetBy(5,5);

tabView = new BTabView(r, "tab_view");
tabView->SetViewColor(216,216,216,0);

r = tabView->Bounds();
r.InsetBy(5,5);
r.bottom -= tabView->TabHeight();
tab = new BTab();
tabView->AddTab(new ConstView(r), tab);
tab->SetLabel("Construct");
tab = new BTab();
tabView->AddTab(new DestView(r), tab);
tab->SetLabel("Obliterate");
```

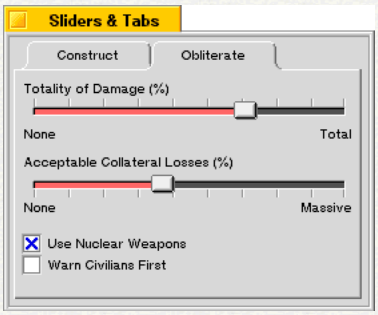
This code creates a BTabView, then sets its view color to the shade of grey used by standard BTabView and BTab objects.

Notice that the frame rectangle for the tabs' target views is being computed by taking the bounds rectangle of the BTabView, insetting it by five pixels on each side (to leave some room between it and the bounding box of the BTabView), and subtracting the tab height from the

bottom edge of the rectangle. This places the target view inside the tab view. You can adjust this computation to position the target view to your liking.

Then two BTab objects are created and added to the BTabView. The ConstView and DestView classes are derived from BView and represent the "Construct" and "Obliterate" panels of the window (including the various controls contained in those panels).

When the user clicks the "Obliterate" tab, the window will change to look like this:



Hook Functions

AddTab()

Can be augmented to assist in setting up tabs as they're added to the BTabView.

Draw()

Draws the BTabView. Can be augmented or reimplemented to customize the appearance of the view.

DrawBox()

Draws the box that surrounds the container view. Can be augmented or reimplemented to customize the appearance of the view.

DrawTabs()

Draws all the tabs in the BTabView. Can be augmented or reimplemented to customize the appearance of the view.

MakeFocus()

Makes the BTabView the focus view, or removes the focus from the BTabView. Can be augmented to take any additional action necessary when this transition occurs.

RemoveTab()

Removes a tab from the BTabView. Can be augmented to perform any additional action necessary.

Select()

Changes the currently-selected tab. Can be augmented to perform any additional action necessary.

SetFocusTab()

Shifts the focus from one tab to another within the BTabView. Can be augmented to perform whatever additional action may be needed when the tab focus changes.

SetTabHeight()

Sets the height of the BTabView's tabs. Can be augmented to perform additional actions.

SetTabWidth()

Sets the width of the tabs. Can be augmented to perform whatever added functions are necessary.

TabAt()

Returns a pointer to a specific BTab object. Can be augmented to perform whatever added functions are necessary.

TabFrame()

Returns the frame rectangle for a given tab. Can be augmented to adjust the frame rectangle for customization of the appearance of your BTabView.

Constructor and Destructor

BTabView()

```
BTabView(BRect frame, const char *name,
    button_width width = B_WIDTH_AS_USUAL,
    uint32 resizingMode = B_FOLLOW_ALL, uint32 flags = B_FULL_UPDATE_ON_RESIZE | B_WILL_DRAW |
B_NAVIGABLE_JUMP | B_FRAME_EVENTS | B_NAVIGABLE)
BTabView(const BTabView &tabView)
BTabView(BMessage *archive)
```

Initializes the BTabView to the *frame* rectangle, stated in its eventual parent's coordinate system, and assigns it the specified *name*, resizing mode, and *flags*. These are described in detail in the BView section.

The *width* parameter, which can be one of the following values, specifies the widths of the tabs in the BTabView. All tabs in a BTabView are the same width.

B_WIDTH_FROM_WIDEST	Each tab's width is determined based upon the width of the widest tab name..
B_WIDTH_AS_USUAL	The default tab width is used for all tabs.
B_WIDTH_FROM_LABEL	The tab width for each tab is based on the width of the tab's label.

The container view for the BTabView's subviews is created and added to the BTabView as well.

~BTabView()

```
virtual ~BTabView()
```

Frees all memory the BTabView allocated, and deletes every BTab attached to it.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*****)**

Returns a new BTabView object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the message doesn't contain archived data for a BTabView, **`Instantiate()`** returns `NULL`.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

Member Functions

AddTab() , RemoveTab()

virtual void **AddTab(**BView **target*, BTab **tab* = **NULL)**

virtual BTab *RemoveTab(int32 tab_index)

`AddTab()` adds the specified *tab* as the rightmost tab in the BTabView. The new tab's target view is set to *target*.

If *tab* is **NULL**, a new BTab object is constructed and added to the BTabView. You can get a pointer to the new tab using the **TabAt()** function

If you choose to reimplement `AddTab()`, you should call the inherited form of this function once the BTab has been customized.

`RemoveTab()` removes the tab with the specified index number from the BTabView and returns a pointer to the BTab object. The BTab is not deleted—if you don't need it anymore, you can do that yourself.

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()** and stores the BTabView in the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Calls **BView::AttachedToWindow()**, then selects the first tab in the tab view.

See also: **BView::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the tabs and the box that encloses the container view that displays the selected tab's target view.

DrawBox()

virtual void **DrawBox(**BRect *selTabRect*)

Draws the box that encloses the container view. *selTabRect* is the frame rectangle of the currently-selected tab; this information is used to allow the box to attach properly to the current tab. This is the same rectangle that the **DrawTabs()** function returns.

This is called for you by the **Draw()** function and is provided primarily as a hook for customizing the appearance of your BTabView.

DrawTabs()

virtual BRect **DrawTabs(**void**)**

Draws all the tabs in the BTabView and returns the frame rectangle of the currently-selected tab. This rectangle should then be passed to **DrawBox()** to draw the container view's enclosing box.

This is called for you by the **Draw()** function and is provided primarily as a hook for customizing the appearance of your BTabView.

FocusTab() see SetFocusTab()

KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes*)

The `KeyDown()` function handles keyboard navigation of the BTabView; the down and left arrow keys move the focus to the left, and the up and right arrow keys move the focus to the right. The space bar and enter keys select the focused tab.

All other keys are passed through to the **BView::KeyDown()** function for further processing.

See also: the **Keyboard Information** appendix, "B_KEY_DOWN" in the **Message Protocols** appendix, **BView::KeyDown()**, **modifiers()**

MakeFocus()

virtual void **MakeFocus(**bool *focused* = true**)**

Makes the BTabView the current focus by first calling **BView::MakeFocus()**, then making the currently-selected tab the focus of the BTabView.

See also: `BView::MakeFocus()`

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Identifies which tab (if any) the user clicked and selects that tab. If the mouse was not inside a tab when clicked, the **`BView::MouseDown`** function is called.

See also: "B_MOUSE_DOWN" in the **Message Protocols** appendix, **`GetMouse()`**

RemoveTab() see AddTab()

Select() , Selection()

virtual void **Select(**int32 *tab***)**
int32 **Selection(**void**)** const

`Select()` selects the tab specified by its index number, first deselecting the previously-selected tab.

`Selection()` returns the index number of the currently-selected tab.

SetFocusTab() , FocusTab()

virtual void **SetFocusTab(**int32 *tab*, bool *focused***)**
int32 **FocusTab(**void**)** const

`SetFocusTab()` sets the *focus* state of the specified *tab*. If *focused* is **`true`**, the specified tab becomes the new focus; if *focused* is **`false`**, the focus is removed from the currently-focused tab but no new focus is set (the BTabView becomes focusless).

`FocusTab()` returns the index number of the tab that is currently the focus.

SetTabHeight() , TabHeight()

virtual void **SetTabHeight(**float *height***)**
float **TabHeight(**void**)** const

`SetTabHeight()` sets the height of the tabs in the BTabView. `TabHeight()` returns the current tab height.

When you change the tab height, the container view for the target views is resized so that the BTabView doesn't change size. Making the tabs taller by N pixelscauses the container view's top edge to move down by N pixels, and decreasing the heights of the tabs increases the height of the container view.

SetTabWidth() , TabWidth()

virtual void **SetTabWidth(**button_width *width***)**
button_width **TabWidth(**void**)** const

`SetTabWidth()` sets the width of the tabs in the BTabView, and `TabWidth()` returns the current width setting. *width* must be one of the following values:

B_WIDTH_FROM_WIDEST	Each tab's width is determined based upon the width of the widest tab name..
B_WIDTH_AS_USUAL	The default tab width is used for all tabs.

TabAt()

virtual BTab ***TabAt(**int32 *tab_index***)**

Returns a pointer to the BTab object at the specified index. The leftmost tab is index number 0.

TabFrame()

virtual BRect **TabFrame(**int32 *tab_index***)**

Returns the frame rectangle of the tab whose index number is specified. The leftmost tab is index number 0.

See also: **`DrawTabs()`**, **`DrawBox()`**, **`Draw()`**

TabHeight() see SetTabHeight()

TabWidth() see SetTabWidth()

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Calls the inherited version of `WindowActivated()`, then calls **`DrawTabs()`** to redraw the tabs.

See also: `BWindow::WindowActivated()`

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_high"	`B_FLOAT_TYPE`	The height of the tabs.
"_but_width"	`B_INT16_TYPE`	The tab width value.
"_sel"	`B_INT32_TYPE`	Which tab is selected.

The archived tabs are added to the "_l_items" field (deep copy only). The tabs' target views are added to the "_views" field (deep copy only), with the leftmost tab inserted first in the "_views" array, and the rightmost tab inserted last.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTabView Function Summary

Function Summary

Function	Description
AddTab()	Adds a tab to the object
Archive()	Archives the BTabView into a BMessage
AttachedToWindow()	Provided for implementation
Draw()	Provided for implementation
DrawBox()	Draws the box enclosing the container view
DrawTabs()	Draws all the tabs
FocusTab()	Returns the index of the focused tab
Instantiate()	Creates a BTabView from a BMessage
KeyDown()	Provided for implementation
MakeFocus()	Provided for implementation
MouseDown()	Provided for implementation
RemoveTab()	Removes a tab from the object
Select()	Selects the specified tab
Selection()	Returns the selected tab
SetFocusTab()	Sets the focus of the specified tab
SetTabHeight()	Sets the height of the tabs
SetTabWidth()	Sets the width of the tabs
TabAt()	Returns a pointer to a given tab
TabFrame()	Returns the frame rectangle of a given tab
TabHeight()	Returns the height of the tabs

BTabView()	Constructor
~BTabView()	Destructor
TabWidth()	Returns the width of the tabs
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

View Hierarchy Functions

Each window (BWindow object) in a BeOS application has an associated tree of BView objects; these BView objects may be contained directly in the BWindow, or they may be contained in other BView objects which belong to the window, hence the tree structure. The functions described in this section are used to construct and use these trees of BView instances. You might want to also read up on the BWindow::AddChild() function, which works much the same as the BView::AddChild() function described below, but which adds a view to a window instead of to another view.

A number of view-hierarchy-related functions can be found in Hook Functions.

AddChild() , RemoveChild()

void **AddChild(**BView **aView*, BView **sibling* = NULL**)**
bool **RemoveChild(**BView **aView***)**

`AddChild()` makes *aView* a child of the BView, provided that *aView* doesn't already have a parent. The new child is added to the BView's list of children immediately before the named *sibling* BView. If the sibling is `NULL` (as it is by default), *aView* isn't added in front of any other view—in other words, it's added to the end of the list. If the BView is attached to a window, *aView* and all its descendants become attached to the same window. Each of them is notified of this change through **`AttachedToWindow()`** and **`AllAttached()`** function calls.

`AddChild()` fails if *aView* already belongs to a view hierarchy. A view can live with only one parent at a time. It also fails if *sibling* is not already a child of the BView.

`RemoveChild()` severs the link between the BView and *aView*, so that *aView* is no longer a child of the BView; *aView* retains all its own children and descendants, but they become an isolated fragment of a view hierarchy, unattached to a window. Each removed view is notified of this change through **`DetachedFromWindow()`** and **`AllDetached()`** function calls.

A BView must be removed from a window before it can be destroyed.

If it succeeds in removing *aView*, **`RemoveChild()`** returns `true`. If it fails, it returns `false`. It will fail if *aView* is not, in fact, a current child of the BView.

When a BView object becomes attached to a BWindow, two other connections are automatically established for it:

- The view is added to the BWindow's flat list of BHandler objects, making it an eligible target for messages the BWindow dispatches.
- The BView's parent view becomes its next handler. Messages that the BView doesn't recognize will be passed to its parent.

Removing a BView from a window's view hierarchy also removes it from the BWindow's flat list of BHandler objects; the BView will no longer be eligible to handle messages dispatched by the BWindow.

See also: `BWindow::AddChild()`, `BLooper::AddHandler()`, `BHandler::SetNextHandler()`, `RemoveSelf()`, `AttachedToWindow()`, `DetachedFromWindow()`

ChildAt() see Parent()

CountChildren() see Parent()

FindView()

BView ***FindView**(const char **name*) const

Returns the BView identified by *name*, or `NULL` if the view can't be found. Names are assigned by the BView constructor and can be modified by the `SetName()` function inherited from BHandler.

`FindView()` begins the search by checking whether the BView's name matches *name*. If not, it continues to search down the view hierarchy, among the BView's children and more distant descendants. To search the entire view hierarchy, use the BWindow version of this function.

See also: `BWindow::FindView()`, `BHandler::SetName()`

MoveBy() , MoveTo()

void **MoveBy**(float *horizontal*, float *vertical*)

void **MoveTo**(BPoint *point*)
void **MoveTo**(float *x*, float *y*)

These functions move the view in its parent's coordinate system without altering its size.

`MoveBy()` adds *horizontal* coordinate units to the left and right components of the frame rectangle and *vertical* units to the top and bottom components. If *horizontal* and *vertical* are positive, the view moves downward and to the right. If they're negative, it moves upward and to the left.

`MoveTo()` moves the upper left corner of the view to *point*—or to (*x*, *y*)—in the parent view's coordinate system and adjusts all coordinates in the frame rectangle accordingly.

Neither function alters the BView's bounds rectangle or coordinate system.

None of the values passed to these functions should specify fractional coordinates; the sides of a view must line up on screen pixels. Fractional values will be rounded to the closest whole number.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new location. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame rectangle.

See also: `FrameMoved()`, `ResizeBy()`, `Frame()`

NextSibling() see Parent()

Parent() , NextSibling() , PreviousSibling() , ChildAt() , CountChildren()

BView ***Parent**(void) const
BView ***NextSibling**(void) const
BView ***PreviousSibling**(void) const
BView ***ChildAt**(int32 *index*) const
int32 **CountChildren**(void) const

These functions provide various ways of navigating the view hierarchy. `Parent()` returns the BView's parent view, unless the parent is the top view of the window, in which case it returns `NULL`. It also returns `NULL` if the BView doesn't belong to a view hierarchy and has no parent.

All the children of the same parent are arranged in a linked list. `NextSibling()` returns the next sibling of the BView in the list, or `NULL` if the BView is the last child of its parent. `PreviousSibling()` returns the previous sibling of the BView, or `NULL` if the BView is the first child of its parent.

`ChildAt()` returns the view at *index* in the list of the BView's children, or `NULL` if the BView has no such child. Indices begin at 0 and there are no gaps in the list. `CountChildren()` returns the number of children the BView has. If the BView has no children, `CountChildren()` returns `NULL`, as will `ChildAt()` for all indices, including 0.

To scan the list of a BView's children, you can increment the index passed to `ChildAt()` until it returns `NULL`. However, it's more efficient to ask for the first child and then use `NextSibling()` to walk down the sibling list. For example:

```
BView *child;
if ( child = myView->ChildAt(0) ) {
   while ( child ) {
      . . .
      child = child->NextSibling();
   }
}
```

PreviousSibling() see Parent()

See also: `AddChild()`

RemoveChild() see AddChild()

RemoveSelf()

bool **RemoveSelf**(void)

Removes the BView from its parent and returns `true`, or returns `false` if the BView doesn't have a parent or for some reason can't be removed from the view hierarchy.

This function acts just like `RemoveChild()`, except that it removes the BView itself rather than one of its children.

See also: `AddChild()`

The Be Book

,

...in lovely HTML...

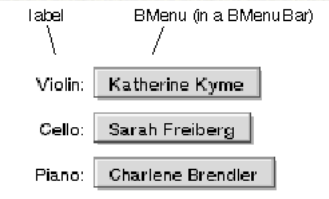
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuField

Derived from: BView
Declared in: be/interface/MenuField.h
Library: libbe.so
Summary: *more...*

A BMenuField object displays a labeled pop-up menu. It's a simple object that employs a BMenuBar object to control a BMenu. All it adds to what a BMenuBar can do on its own is a label and a more control-like user interface that includes keyboard navigation.



The functions defined in this class resemble those of a BControl (`SetLabel()`, `IsEnabled()`), especially a BTextControl (`SetDivider()`, `Alignment()`). However, unlike a real BControl object, a BMenuField doesn't maintain a current value and it can't be invoked to send a message. All the control work is done by items in the BMenu.

Constructor and Destructor

BMenuField()

```
BMenuField(BRect frame, const char *name,
    const char *label,
    BMenu *menu,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BMenuField(BRect frame, const char *name,
    const char *label,
    BMenu *menu,
    bool fixedSize,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BMenuField(BMessage *archive)
```

Initializes the BMenuField object with the specified *frame* rectangle, *name*, *resizingMode*, and *flags*. These arguments are the same as for any BView object and are passed unchanged to the BView constructor. When the object is attached to a window, the height of its frame rectangle will be adjusted to fit the height of the text it displays, which depends on the user's preferred font for menus.

By default, the frame rectangle is divided horizontally in half, with the *label* displayed on the left and the *menu* on the right. This division can be changed with the **SetDivider()** function. The *menu* is assigned to a BMenuBar object and will pop up under the user's control. For most uses, the *menu* should be a BPopUpMenu object.

The second form of the constructor accepts an added argument, *fixedSize*. If this is **true**, the BMenuField won't adjust its size based on the width of the items in the menu.

~BMenuField()

```
virtual ~BMenuField()
```

Frees the label, the BMenuBar object, and other memory allocated by the BMenuField.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BMenuField object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BMenuField object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Alignment() see SetAlignment()

Archive()

```
virtual status_t Archive(BMessage *archive, bool deep = true) const
```

Calls the inherited version of `Archive()`, which will, in the normal course of things, archive the child BMenuBar and the BMenu it displays, provided the *deep* flag is `true`. This function then adds the label, divider, and current state of the BMenuField to the BMessage *archive*.

See also: `BArchivable::Archive()`, `Instantiate()` static function

AttachedToWindow() , AllAttached()

```
virtual void AttachedToWindow(void)
virtual void AllAttached(void)
```

These functions override their BView counterparts to make the BMenuField's background color match the color of its parent view and to adjust the height of the view to the height of the BMenuBar child it contains. The height of the child depends on the size of the user's preferred font for menus.

See also: `BView::AttachedToWindow()`

Divider() see SetDivider()

Draw()

```
virtual void Draw(BRect updateRect)
```

Overrides the BView version of this function to draw the view's border and label. The way the menu field is drawn depends on whether it's enabled or disabled and whether or not it's the current focus for keyboard actions.

See also: `BView::Draw()`

IsEnabled() see SetEnabled()

KeyDown()

virtual void **KeyDown(**const char **bytes*, int32 *numBytes***)**

Augments the BView version of **KeyDown()** to permit keyboard navigation to and from the view and to allow users to open the menu by pressing the space bar.

See also: **BView::KeyDown()**

Label() see SetLabel()

MakeFocus()

virtual void **MakeFocus(**bool *focused***)**

Augments the BView version of **MakeFocus()** to enable keyboard navigation. This function calls **Draw()** when the BMenuField becomes the focus view and when it loses that status.

See also: **BView::MakeFocus()**

Menu() , MenuBar() , MenuItem()

BMenu ***Menu(**void**)** const

BMenuBar ***MenuBar(**void**)** const

BMenuItem ***MenuItem(**void**)** const

Menu() returns the BMenu object that pops up when the user operates the BMenuField; **MenuBar()** returns the BMenuBar object that contains the menu. The BMenuBar is created by the BMenuField; the menu is assigned to it during construction. `MenuItem()` returns the first BMenuItem assigned to the BMenuBar object containing the menu.

See also: the BMenuField constructor

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **MouseDown()** to enable users to pop up the menu using the mouse, even if the cursor isn't directly over the menu portion of the bounds rectangle.

See also: **BView::MouseDown()**

SetAlignment() , Alignment()

virtual void **SetAlignment(**alignment *label***)**
alignment **Alignment(**void**)** const

These functions set and return the alignment of the label in its portion of the frame rectangle.

B_ALIGN_LEFT	The label is aligned at the left side of the bounds rectangle.
B_ALIGN_RIGH T	The label is aligned at the right boundary of its portion of the bounds rectangle.
B_ALIGN_CEN TER	The label is centered in its portion of the bounds rectangle.

The default is `B_ALIGN_LEFT`.

SetDivider() , Divider()

virtual void **SetDivider(**float *xCoordinate*)
float **Divider(**void**)** const

These functions set and return the *x* coordinate value that divides the bounds rectangle between the label's portion on the left and the portion that holds the menu on the right. The coordinate is expressed in the BMenuField's coordinate system.

The default divider splits the bounds rectangle in two equal sections. By resetting it, you can provide more or less room for the label or the menu.

SetEnabled() , IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**
bool **IsEnabled(**void**)** const

`SetEnabled()` enables the BMenuField if the *enabled* flag is `true`, and disables it if the flag is `false`. `IsEnabled()` returns whether or not the object is currently enabled. When disabled, the BMenuField doesn't respond to mouse and keyboard manipulations.

If the *enabled* flag changes the current state of the object, `SetEnabled()` causes the view to be redrawn, so that its new state can be displayed to the user.

SetLabel() , Label()

virtual void **SetLabel(**const char **string***)
const char ***Label(**void**)** const

`SetLabel()` frees the current label and, if the argument it's passed is not `NULL`, replaces it with a copy of *string*. `Label()` returns the current label. The string it returns belongs to the

BMenuField object.

See also: the BMenuField constructor

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Makes sure that the BMenuField is redrawn when the window is activated and deactivated, provided that it's the current focus view.

See also: **`BView::WindowActivated()`**

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_label"	`B_STRING_TYPE`	BMenuField label.
"_disable"	`B_BOOL_TYPE`	Exists and `true` only if menu is disabled.
"_align"	`B_INT32_TYPE`	Alignment of label in the frame rectangle.
"_divide"	`B_FLOAT_TYPE`	The x-coordinate dividing the bounds rectangle between the label and the menu.

The following views are added to the "_views" field (deep copy only):

Name	Level	Meaning
"_mc_mb_"	0	Private menu bar.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMenuField Function Summary

Function Summary

Function	Description
Alignment()	Returns alignment of label in frame
AllAttached()	Provided for implementation
Archive()	Archives a BMenuField into a BMessage
AttachedToWindow()	Provided for implementation
Divider()	Returns coordinate dividing label and menu
Draw()	Provided for implementation
Instantiate()	Creates a BMenuField from a BMessage
IsEnabled()	Tests whether the BMenuField is enabled
KeyDown()	Provided for implementation
Label()	Returns BMenuField label
MakeFocus()	Provided for implementation
Menu()	Returns BMenu controlled by object
MenuBar()	Returns BMenuBar created by the object
BMenuField()	Constructor
~BMenuField()	Destructor
MenuItem()	Returns first BMenuItem for BMenuBar created by the object
MouseDown()	Provided for implementation

SetAlignment()	Sets alignment of label in frame
SetDivider()	Sets coordinate dividing label and menu
SetEnabled()	Enables or disables BMenuField
SetLabel()	Sets BMenuField label
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDragger

Derived from: public BView
Declared in: be/interface/Dragger.h
Library: libbe.so
Summary: *more...*

A BDragger is a view that lets users drag and drop some other view. The other view is the *target* of the BDragger and its immediate relative—a sibling, a parent, or an only child. The BDragger draws a handle, usually at the corner of the target view, that the user can grab. When the user drags the handle the target view appears to move with the handle.

When dragged in this way, the target view itself doesn't actually move. Instead, the view is archived in a BMessage object and the BMessage is dragged. When the BMessage is dropped, the target BView can be reconstructed from the archive (along with the BDragger). The new object is a duplicate—a replicant—of the target view.

This class works closely with the BShelf class. A BShelf object accepts dragged BViews, reconstructs them from their archives, and installs them in another view hierarchy.

BDraggers are under the control of DeskBar's "Show Replicants" / "Hide Replicants" menu item. Showing replicants means that the BDragger handles are visible on-screen; hiding replicants means that the handles are hidden.

Constructor and Destructor

BDragger()

```
BDragger(BRect frame, BView *target, uint32 resizingMode = B_FOLLOW_NONE,
uint32 flags = B_WILL_DRAW)
BDragger(BMessage *archive)
```

Creates a new BDragger and sets its target view. The BDragger and the target BView must be directly related in the view hierarchy (as parent-child or as siblings); but, note well, the constructor doesn't establish this relationship for you. After you construct you BDragger, you have to do one of three things:

- add the target as a child of the dragger,
- add the dragger as a child of the target,
- or add the dragger as a sibling of the target.

If you add the target as a child of BDragger, it should be the only child that the BDragger has.

A BDragger draws in the right bottom corner of its *frame* rectangle. If the target view is a parent or a sibling of the BDragger, that rectangle needs to be no larger than the image the BDragger draws (the handle). However, if the target is the BDragger's child, the dragger's frame rectangle must enclose the target's frame (so that the dragger doesn't clip the target).

A BDragger is fully functional once it has been constructed and attached to the view hierarchy of its target. You don't need to call any other functions. However, the whole endeavor fails if the target BView can't be archived.

~BDragger()

```
virtual ~BDragger()
```

Frees all memory the BDragger allocated (principally for the bitmap image it draws).

Static Functions

HideAllDraggers() , ShowAllDraggers() , AreDraggersDrawn()

static status_t **HideAllDraggers**(void)
static status_t **ShowAllDraggers**(void)
static bool **AreDraggersDrawn**(void)

These functions communicate with all BDragger objects in all applications (provided they're attached to windows). `HideAllDraggers()` hides the BDragger objects so that they're not visible on-screen. `ShowAllDraggers()` undoes the effect of `HideAllDraggers()` and causes all BDragger objects to draw their handles. The Show Replicants / Hide Replicants menu item does its work through these functions.

`HideAllDraggers()` may or may not hide the BDragger view in the way that BView's `Hide()` function does. The BDragger may still be visible, although it won't draw anything until `ShowAllDraggers()` is called. Therefore, if the target BView is the BDragger's child, it will not be hidden when `HideAllDraggers()` erases its parent.

`AreDraggersDrawn()` returns `true` when the BDraggers are shown and `false` when they're hidden.

Instantiate()

static BArchivable ***Instantiate**(BMessage ****archive**)

Returns a new BDragger object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. If the *archive* message doesn't contain and archived BDragger, `Instantiate()` returns `NULL`.

ShowAllDraggers() see HideAllDraggers()

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Records the BDragger's hierarchical relationship to the target view and then calls **BView::Archive()**. The *deep* flag has no significance for BDragger itself, but note that the flag is passed on to the BView version.

AttachedToWindow() , DetachedFromWindow()

virtual void **AttachedToWindow(**void**)**

virtual void **DetachedFromWindow(**void**)**

AttachedToWindow() makes sure that the BDragger is under the control of the **HideAllDraggers()** and **ShowAllDraggers()** functions, makes its low and background view colors match the view color of its parent, and determines the BDragger's precise relationship to its target view. To make this determination, the target must be in the view hierarchy; it can't be added to the window after the BDragger is. For example, if the target is the BDragger's child, it should be added to the BDragger and then the BDragger added to the window.

DetachedFromWindow() removes the BDragger from the control of the **HideAllDraggers()** and **ShowAllDraggers()** functions.

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the handle—or fails to draw it and has the parent view draw in that area instead, if all BDraggers are hidden.

IsVisibilityChanging()

protected: bool **IsVisibilityChanging(**void**)** const

Returns `true` if two things are true:

- The BDragger is the parent of its target.
- The BDragger handle was visible but now should not be, or it wasn't visible and now should be.

Otherwise, this function returns `false`.

What's this function for? It's in the API so derived classes can implement their own versions of **Draw()**. If the BDragger isn't the parent of its target, the visibility of the BDragger view can be controlled by the **Hide()** and **Show()** functions rather than **Draw()**.

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Responds to messages that regulate the visibility of the BDragger handle.

MouseDown()

virtual void **MouseDown(**BPoint *where***)**

Responds to a **B_MOUSE_DOWN** message by archiving the target view (and the BDragger) and initiating a drag-and-drop operation, or by taking other appropriate action.

PopUp() , SetPopUp()

BPopUpMenu ***PopUp(**void**)** const

status_t **SetPopUp(**BPopUpMenu **context_menu***)**

Returns and sets the BPopUpMenu displayed when the user right clicks on the BDragger view after it has been attached to a BShelf.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_popup"	`B_MESSAGE_TYPE`	Archive of the pop-up menu. Field present only if this menu was set with `SetPopUp()`.
"_rel"	`B_INT32_TYPE`	Relationship to the target view; one of: `TARGET_IS_PARENT`, `TARGET_IS_CHILD`, or `TARGET_IS_SIBLING`.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDragger Function Summary

Function Summary

Function	Description
Archive()	Archives a BDragger into a BMessage
AreDraggersDrawn()	Tests whether BDraggers are displayed
AttachedToWindow()	Provided for implementation
DetachedFromWindow()	Provided for implementation
BDragger()	Constructor
~BDragger()	Destructor
Draw()	Provided for implementation
HideAllDraggers()	Hides all BDraggers in all applications
Instantiate()	Creates a BDragger from a BMessage
IsVisibilityChanging()	protected: Tests whether the BDragger's visibility is changing
MessageReceived()	Provided for implementation
MouseDown()	Provided for implementation
PopUp()	Returns BPopUpMenu displayed when handle is clicked
SetPopUp()	Sets BPopUpMenu displayed when handle is clicked
ShowAllDraggers()	Unhides all BDraggers in all applications

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShelf

Derived from: public BHandler
Declared in: be/interface/Shelf.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BShelf is an object that you "attach" to a view in order to make the view accept dropped BDragger objects. In user-talk, a shelf receives and displays replicants. Attaching a BShelf to a view (called the "container" view) is remarkably simple:

```
BShelf *shelf = new BShelf(container_view);
```

That's all there is to it: With this single line of code, `container_view` is primed to accept and (automatically) display dropped replicants. A dropped replicant becomes a child of the container view. The container view itself can be *any* BView object—you don't alter the view in any way, or even tell it that it's going to be a container.

Attaching a shelf to a view is performed by the BShelf constructor only. You can't create a BShelf and *then* decide which view you want it to serve.

Dropping into the View Hierarchy

When the user drops a replicant on a container view, the view receives a `B_ARCHIVED_OBJECT` message that contains a BDragger and the dragger's target (a BView). These two views (the BDragger and its target) are related as parent-child, child-parent, or as siblings (as explained in the BDragger class). The "more elderly" of the two objects is added as a child of the container view; if they're siblings, the two objects are both added as children.

You can also send or post `B_ARCHIVED_OBJECT` messages to a BShelf to simulate a drag and drop.

Other BShelf Features

- **Settings**. A BShelf can configure itself from a "settings" file, and can write its contents to that file. Your only chance to associate a BShelf with a settings file is during construction. You can save the contents at any time through the `Save()` function.
- **Replicant rejection**. A BShelf can reject replicants, and can adjust the position of the replicants that it accepts. These features are provided through the hook functions `CanAcceptReplicantMessage()`, `CanAcceptReplicantView()`, and `AdjustReplicantBy()`. A named shelf can also reject replicants, as explained next.

- **BShelf names**. When a replicant message is dropped on the shelf, the shelf's name (set in the constructor) is compared to the dropped message's "shelf_type" field (if it has one). If the two don't match, the replicant rejects the shelf. In this way, individual replicants can be picky about the shelves that they want to be displayed on. By default, a shelf accepts replicants that don't have a "shelf_type" field. However, if a BShelf is set to "enforce type" (set through `SetTypeEnforced()`), then it will only accept replicants that name the shelf (in their "shelf_type" fields).
- A BShelf assigns each replicant a unique id that's valid for as long as the replicant is attached to the shelf. Replicant ids don't change, even if the shelf is `Save()`'d and then later restored (through the constructor).

Hook Functions

CanAcceptReplicantMessage()
Invoked when a replicant BMessage is received by the BShelf. A return value of **false** rejects the replicant.

CanAcceptReplicantView()
Invoked after the message has been accepted (by the above); this is the shelf's chance to reject on the basis of the view that the message contains. A return value of **false** rejects the replicant.

AdjustReplicantBy()
Invoked after the replicant has been accepted, but before it's displayed. The function can return a BPoint that offsets the replicant's frame.

Constructor and Destructor

BShelf()

```
BShelf(BView *view, bool allowsDragging = true, const char *name = NULL)
BShelf(const entry_ref *ref, BView *view, bool allowsDragging = true,
    const char *name = NULL)
BShelf(BDataIO *stream, BView *view, bool allowsDragging = true,
    const char *name = NULL)
```

Initializes the BShelf object so that it serves a container *view*. The versions that accept an **entry_ref** or BDataIO argument prime the shelf so that it (initially) contains the replicants that are archived in the referred to file or stream using **Save()**. The *ref*/*stream* argument is also used as the archival repository when you tell your BShelf to **Save()** itself.

If the *allowsDragging* flag is `true`, the user will be able to drag replicant view within the container's bounds. If the flag is `false`, dropped views stay where they're first put.

name is the BShelf's handler name. The name can be important: It's compared to the replicant's "shelf_type" field, as explained in **AddReplicant()**.

There's an archive-accepting version of the BShelf constructor declared in **Shelf.h**. Don't use it.

~BShelf()

```
virtual ~BShelf()
```

The destructor calls **Save()**, and then frees the object.

Static Functions

Instantiate()

Instantiate() is currently a no-op that returns **NULL**. You can't archive a BShelf. If you want to archive something, archive the shelf's contents by calling **Save()**.

Member Functions

AddReplicant()

status_t **AddReplicant(**BMessage **archive*, BPoint *point***)**

This function is invoked automatically when a replicant is dropped on the BShelf. The *archive* message contains the BDragger archive that's being dropped; *point* is where, within the container view's bounds, the message was dropped. The function goes through these steps to reject and adjust the replicant:

- First, it invokes the **`CanAcceptReplicantMessage()`** hook function. If the hook returns **`false`**, then **`AddReplicant()`** doesn't add the replicant.
- Next, it looks for a "shelf_type" string field in the BMessage. If it finds one and the value of the field doesn't match the BShelf's name, the replicant is rejected.
- If type enforcement is `true` (see **`SetTypeEnforced()`**) and the shelf has a name, then the BMessage must have a "shelf_type" string and this string must match the shelf name. Otherwise, the replicant is rejected.
- There's no specific API for adding the "shelf_type" field to a view. If you want to configure your views to accept only certain BShelf objects, you have to add the field directly as part of the view's **`Archive()`** implementation.
- The archive message is then unarchived (the replicant is instantiated). If the archive doesn't contain a BView, the message is passed on to another handler (**`B_DISPATCH_MESSAGE`** is returned).
- **`CanAcceptReplicantView()`** hook function is called next (with a return of **`false`** meaning rejection).
- Finally, **`AdjustReplicantBy()`** is called, and the replicant is drawn in the container view.

Except in the case of a no-view archive, **`AddReplicant()`** returns **`B_SKIP_MESSAGE`**.

If you want the ensure that the replicant is unique within the container view, add a "be:unique_replicant" entry of type **`B_BOOL_TYPE`** to the archive with the value **`true`**.

It's possible to archive a BDragger and call this function yourself, although that's not its expected use.

AdjustReplicantBy()

protected: virtual BPoint **AdjustReplicantBy(**BRect *destRect*, BMessage **archive***)** const

This hook function is invoked automatically when a replicant is dropped on the BShelf. It gives the shelf a chance to fine-tune the placement of the BDragger and its target view.

destRect is the rectangle (in the container view's coordinates) in which the dropped replicant is about to be drawn. Exactly what the rectangle means depends on the relationship between the dragger and its target:

- If the dragger is the target's parent, then *destRect* encloses the BDragger's frame.
- Otherwise (if the target is the parent, or if the two views are siblings), *destRect* encloses the target's frame. Note that in the case of siblings, the BDragger's frame isn't included in the rectangle.

archive is the archive message that was dropped on the shelf.

The BPoint that this function returns offsets (is added into) the location of the replicant. If you don't want to move the replicant, return BPoint(0, 0). Note that the BDragger and the view are *both* moved by this offset, even in the case where *destRect* doesn't include the dragger's frame.

This function ignores the "allows dragging" flag given in the BShelf constructor. In other words, you can adjust a replicant's placement through this function even if the BShelf doesn't otherwise allow dragging.

AllowsDragging() see SetAllowsDragging()
AllowsZombies() see SetAllowsZombies()

Archive()

Archive() is currently a no-op that returns **B_ERROR**. You can't archive a BShelf. If you want to archive something, archive the shelf's contents by calling **Save()**.

CanAcceptReplicantMessage() , CanAcceptReplicantView()

protected: virtual bool **CanAcceptReplicantMessage(**BMessage **archive***)** const

protected: virtual bool **CanAcceptReplicantView(**BRect *destRect*,
BView **view*,
BMessage **archive***)** const

These hook function are invoked from within **AddReplicant()** whenever a replicant is dropped on the BShelf. You can implement these functions to reject unwanted replicants.

CanAcceptReplicantMessage() is called first; the argument is the archive that (should) contain the replicated view. If you don't like the look of the archive, return **false** and the message will be thrown away. Note that you shouldn't return **false** if the archive doesn't seem to be in the

correct form (specifically, if it doesn't contain any views). Rejection of such messages is handled more elegantly (and after this function is invoked) by the **AddReplicant()** function.

CanAcceptReplicantView() is invoked after the message has been unarchived. *destRect* is the rectangle that the replicant will occupy in the BShelf's container view's coordinates. *view* is the replicated view itself. *archive* is theoriginal message.

If either function returns **false**, the replicant is rejected and the message is thrown away (it isn't passed on to another handler). A return of **true** does the obvious thing.

CountReplicants()

```
int32 CountReplicants(void) const
```

Returns the number of replicants attached to the shelf.

DeleteReplicant()

```
status_t DeleteReplicant(BView *replicant_view)
status_t DeleteReplicant(const BMessage *archive)
status_t DeleteReplicant(uint32 uid)
```

Removes the specified replicant from the shelf. It identifies replicants by either a view, a replicant message, or a unique id.

DisplaysZombies() see SetAllowsZombies()

IndexOf() , ReplicantAt()

```
int32 IndexOf(const BView *replicant_view) const
int32 IndexOf(const BMessage *archive) const
int32 IndexOf(uint32 uid) const
```

```
BMessage *ReplicantAt(int32 index, BView **view = NULL, uint32 *uid = NULL,
    status_t *err = NULL) const
```

`IndexOf()` returns the index of a specified replicant in the shelf, or -1 if no such replicant exists. It accepts either a view, a replicant message, or a unique id as identifiers.

`ReplicantAt()` returns information about a replicant in a shelf given its index (as returned by `IndexOf()`). It returns the BMessage archive of the replicant or `NULL` if the index is invalid. It returns the *view* of the replicant as well as its *uid*, if these parameters are non-`NULL`. It returns an error message in *err* if there was an error in initializing the given replicant.

IsDirty() see Save()

- IsTypeEnforced() see SetTypeEnforced()
- ReplicantAt() see IndexOf()

ReplicantDeleted()

```
protected: virtual void ReplicantDeleted(int32 index,
    const BMessage *archive,
    const BView *view)
```

This hook function is invoked from within `DeleteReplicant()` to indicate a replicant has been deleted from the BShelf. It is called after the view has been removed from the shelf with the *index* of the replicant in the shelf, its *archive* message, and the detached *view*.

Save() , SaveLocation() , SetSaveLocation() , SetDirty() , IsDirty()

```
status_t Save(void)
BDataIO *SaveLocation(entry_ref *ref) const
status_t SetSaveLocation(BDataIO *data_io)
status_t SetSaveLocation(const entry_ref *ref)
virtual void SetDirty(bool flag)
bool IsDirty(void)
```

Writes the shelf's contents (the replicants that it contains) as an archive to the `entry_ref` or BDataIO object that was specified in the constructor. You can also set the location where the shelf is saved with `SetSaveLocation()` and fetch it with `SaveLocation()`. The entry_ref is stored in *ref* (if *ref* is non-`NULL`) and the BDataIO the shelf will be written to is returned. If the shelf will be written to an entry_ref that is not a BDataIO, `SaveLocation()` returns `NULL`.

By default, the save is only performed if the object's "dirty" flag is set—in other words, if it has changed since it was last written. You can force set the dirty flag by calling `SetDirty()`.

`IsDirty()` returns the current state of the "dirty" flag.

SetAllowsDragging() , AllowsDragging()

```
void SetAllowsDragging(bool state)
bool AllowsDragging(void) const
```

`SetAllowsDragging()` determines whether the BShelf accepts replicants dragged into the view by the user. `AllowsDragging()` returns whether the BShelf accepts replicants dragged by the user into the container view's frame?

SetAllowsZombies() , AllowsZombies() , SetDisplayZombies() , AllowsZombies()

void **SetAllowsZombies(**bool *state***)**
bool **AllowsZombies(**void**)** const
void **SetDisplaysZombies(**bool *state***)**
bool **DisplaysZombies(**void**)** const

`SetAllowsZombies()` determines whether the BShelf accepts zombie views. `AllowsZombies()` returns whether the BShelf accepts zombie views. A zombie view is one whose associated executable cannot be located.

Similarly, `SetDisplaysZombies()` determines whether the BShelf displays zombie views and `DisplaysZombies()` returns whether the BShelf displays zombie views.

SetDisplaysZombies() see SetAllowsZombies()

SetTypeEnforced() , IsTypeEnforced()

void **SetTypeEnforced(**bool *state***)**
bool **IsTypeEnforced(**void**)** const

These two methods set and return the type enforcement flag. When it is `true`, the shelf compares its name to the "shelf_type" field of any dropped messages. The replicant is accepted only if the two match. If the dropped message does not have a "shelf_type" field, then it is rejected. Type enforcement is `false` by default.

Scripting Support

The BShelf class implements the suite called "suite/vnd.Be-shelf" consisting of the following message:

The Replicant Property

Message	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Returns number of replicants in the shelf.
B_CREATE_PROPERTIES	B_DIRECT_SPECIFIER	Adds the archived replicant in the BMessage "data" to the view at the BPoint in "location."
B_DELETE_PROPERTY	B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER, B_NAME_SPECIFIER, B_ID_SPECIFIER	Removes the specified replicant from the shelf.
B_GET_PROPERTY	B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER, B_NAME_SPECIFIER, B_ID_SPECIFIER	Archives the specified replicant into a BMessage in "result."
anything else	B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER, B_NAME_SPECIFIER, B_ID_SPECIFIER	Directs the scripting message to the replicant interface "suite/vnd.Be-replicant" (described below), first popping the current specifier off the stack.

Accesses replicants contained in the BShelf **The "Replicant" property provides access to the replicants contained in a BShelf. It also allows you to manipulate the replicants themselves by forwarding certain messages to a "suite/vnd.Be-replicant" interface. This interface consists of the following messages:** **The ID Property**	**Message**	**Specifiers**	**Meaning**
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the replicant ID.	

The replicant ID **The Name Property**	**Message**	**Specifiers**	**Meaning**
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the name of the replicant view.	

The name of the replicant view **The Signature Property**	Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the signature of the add-on containing the code for the replicant.	

The replicant add-on MIME signature **The Suites Property**	Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns "suite/vnd.Be-replicant" in "suites" and a flattened BPropertyInfo describing the scripting suite in "messages."	

The supported scripting suites **The View Property**	Message	Specifiers	Meaning
any	B_DIRECT_SPECIFIER	Directs the scripting message to the replicant view, first popping the current specifier off the specifier stack.	

Redirects messages to the replicant view

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShelf Function Summary

Function Summary

Function	Description
AddReplicant()	Adds a replicant to the BShelf
AdjustReplicantBy()	Repositions a replicant view
AllowsDragging()	Tests whether the BShelf accepts replicants dropped by the user
AllowsZombies()	Tests whether the BShelf displays zombie views
Archive()	Use **`BShelf::Save()`** instead
CanAcceptReplicantMessage()	Hook called when a replicant is dropped on the BShelf
CountReplicants()	Returns the number of replicants attached to the BShelf
DeleteReplicant()	Removes a replicant from the BShelf
IndexOf()	Returns the index of a specified replicant in the BShelf
Instantiate()	Use **`BShelf::Save()`** instead
IsDirty()	Tests if a change has been made in the BShelf
IsTypeEnforced()	Tests the type enforcement flag

ReplicantAt()	Returns the replicant at a given index in the BShelf
ReplicantDeleted()	Hook called when a replicant is removed from the BShelf
Save()	Writes the BShelf's contents to an archive
SaveLocation()	Returns the location where the contents of the BShelf are saved
SetAllowsDragging()	Determines whether the BShelf accepts replicants dropped by the user
SetAllowsZombies()	Determines whether the BShelf accepts zombie views
SetDirty()	Marks a change in the BShelf
SetDisplayZombies()	Determines whether the BShelf displays zombie views
SetSaveLocation()	Sets the location where the contents of the BShelf are saved
SetTypeEnforced()	Sets the type enforcement flag
BShelf()	Constructor
~BShelf()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDataIO

Derived from: none
Declared in: be/support/DataIO.h
Library: libbe.so
Summary: *more...*

BDataIO is an abstract class that defines protocols for objects that can `Read()` and `Write()` data. BPositionIO, BDataIO's main subclass, declares an additional set of functions (`ReadAt()`, `WriteAt()`, `Seek()`, and `Position()`) for objects that can keep track of the current position in the I/O buffer; it implements `Read()` and `Write()` in terms of these other functions.

Neither BDataIO nor BPositionIO declares any data members, nor do they implement the functions in the protocols they declare. It's up to derived classes to implement them based on the properties of the particular kinds of data sources/repositories they represent.

Constructor and Destructor

BDataIO()

```
BDataIO(void)
```

Does nothing.

~BDataIO()

```
virtual ~BDataIO()
```

Does nothing.

Member Functions

Read() , ReadAt()

```
virtual ssize_t Read(void *buffer, size_t numBytes) = 0
```

Read() is implemented by derived classes to copy *numBytes* bytes of data from the object to the *buffer*. It should return the number of bytes actually read, which may be 0, or an error code if something goes wrong.

Write()

```
virtual ssize_t Write(const void *buffer, size_t numBytes) = 0
```

Write() is implemented by derived classes to copy *numBytes* bytes of data from the *buffer* to the object. It should return the number of bytes actually written, which may be 0, or an error code if the operation fails.

The Be Book The Support Kit The Support Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDataIO Function Summary

Function Summary

Function	Description
BDataIO()	Constructor
~BDataIO()	Destructor
Read()	Overridden to read data from the current position
Write()	Overridden to write data to the current position

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPositionIO

Derived from: BDataIO
Declared in: be/support/DataIO.h
Library: libbe.so
Summary: *more...*

BPositionIO is an abstract class that defines protocols for performing input/output operations. See BDataIO for information about this class.

Constructor and Destructor

BPositionIO()

```
BPositionIO(void)
```

Does nothing. Constructors in derived classes should initialize the object to default values—for example, set the current position to the beginning of the data.

~BPositionIO()

```
virtual ~BPositionIO()
```

Does nothing.

Member Functions

Position() see Seek()

Read() , ReadAt()

```
virtual ssize_t Read(void *buffer, size_t numBytes)
virtual ssize_t ReadAt(off_t position,
    void *buffer,
    size_t numBytes) = 0
```

Read() copies *numBytes* bytes of data from the object to the *buffer*. It returns the number of bytes actually read (which may be 0) or an error code if something goes wrong. Read() is implemented using Seek(), ReadAt(), and Position(), all of which are pure virtual functions.

ReadAt() is a pure virtual that must be implemented by derived classes to read *numBytes* bytes of data beginning at *position* in the data source, and to place them in the *buffer*. It should return the number of bytes actually read, or an error code if something goes wrong.

Seek() , Position()

```
virtual off_t Seek(off_t position, int32 mode) = 0
virtual off_t Position(void) const = 0
```

Seek() is implemented by derived classes to modify the current position maintained by the object. The current position is an offset in bytes from the beginning of the object's data. How the *position* argument is interpreted will depend on the *mode* flag. Three possible modes should be supported:

- **SEEK_SET**. The *position* passed is an offset from the beginning of allocated memory; in other words, the current position should be set to *position*.
- **SEEK_CUR**. The *position* argument is an offset from the current position; the current position should be incremented by *position*.
- **SEEK_END**. The *position* argument is an offset from the end of the data; the current position should be the sum of *position* plus the number of bytes in the data.

For the **SEEK_SET** mode, *position* should be a positive value. The other modes permit negative offsets.

Seek() should return the new position.

Position() should return the current position.

SetSize()

```
virtual status_t SetSize(off_t numBytes)
```

Returns B_ERROR to indicate that, in general, BPositionIO objects can't set the amount of memory in the repositories they represent. However, the BMallocIO class in this kit and BFile in the Storage Kit implement SetSize() functions that override this default.

Write() , WriteAt()

```
virtual ssize_t BPositionIO::Write(const void *buffer, size_t numBytes)
virtual ssize_t BPositionIO::WriteAt(off_t position,
    const void *buffer,
    size_t numBytes) = 0
```

Write() copies *numBytes* bytes of data from the object into the *buffer*. It returns the number of bytes actually written (which may be 0) or an error code if something goes wrong. Write() is implemented using Seek(), Position(), and WriteAt(), all of which are pure virtual functions.

WriteAt() is a pure virtual that must be implemented by derived classes to copy *numBytes* bytes of data from *buffer* and write them into the object, starting at byte *position* within the object. It should return the number of bytes actually read, or an error code if something goes wrong.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPositionIO Function Summary

Function Summary

Function	Description
Position()	Overridden to return the current position
BPositionIO()	Constructor
~BPositionIO()	Destructor
ReadAt()	Overridden to read data from a given position
Read()	Overridden to read data from the current position
Seek()	Overridden to move the current position
SetSize()	Overridden to set the size of the data
WriteAt()	Overridden to write data to an arbitrary position
Write()	Overridden to write data to the current position

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMallocIO

Derived from: BPositionIO
Declared in: be/support/DataIO.h
Library: libbe.so
Summary: *more...*

BMallocIO represents a buffer of dynamically allocated memory. A BMallocIO object allocates the buffer when you first call `Write()` or `WriteAt()`. On subsequent calls, BMallocIO makes sure enough memory is allocated to hold all the data you intend to write, reallocating it if necessary. Memory is allocated in multiples of a block size that you can set.

Constructor and Destructor

BMallocIO()

```
BMallocIO(void)
```

The BMallocIO constructor creates an empty object and sets the default block size to 256 bytes. The constructor doesn't allocate any memory; memory is allocated when you first write to the object or when you call **SetSize()** to set the amount of memory.

~BMallocIO()

```
virtual ~BMallocIO()
```

The BMallocIO destructor frees all memory that was allocated by the object.

Member Functions

Buffer() , BufferLength()

const void *__Buffer__(void) const
size_t __BufferLength__(void) const

`Buffer()` returns a pointer to the memory that the BMallocIO object has allocated, or **NULL** if it hasn't yet had occasion to allocate any memory. `BufferLength()` returns the number of data bytes in the buffer (not necessarily the full number of bytes that were allocated).

Position() see Seek()

ReadAt()

virtual ssize_t __ReadAt__(off_t *position*, void **buffer*, size_t *numBytes*)

Reads up to *numBytes* bytes of data from the object and copies it into the *buffer*. Returns the actual number of bytes placed in the buffer. The data is read beginning at the *position* offset.

This function doesn't read beyond the end of the data. If there are fewer than *numBytes* of data available at the *position* offset, it reads only through the last data byte and returns a smaller number than *numBytes*. If *position* is out of range, it returns 0.

Seek() , Position()

virtual off_t __Seek__(off_t *position*, int32 *mode*)
virtual off_t __Position__(void) const

`Seek()` sets the position in the data buffer where the `Read()` and `Write()` functions (inherited from BPositionIO) begin reading and writing. How the *position* argument is understood depends on the *mode* flag. There are three possible modes:

- **SEEK_SET**. The *position* passed is an offset from the beginning of allocated memory; in other words, the current position is set to *position*. For this mode, *position* should be a positive value.
- **SEEK_CUR**. The *position* argument is an offset from the current position; the value of the argument is added to the current position.
- **SEEK_END**. The *position* argument is an offset from the end of the object's data (not necessarily from the end of allocated memory). Positive values seek beyond the end of the data; negative values seek backwards into the data.

Attempts to seek beyond the end of allocated memory are legal: When Write() is subsequently called, the object updates its conception of where the data ends to bring the current position within range. If necessary, enough memory will be allocated to accommodate any data added at the current position.

Both Seek() and Position() return the current position as an offset in bytes from the beginning of allocated memory.

SetBlockSize() , SetSize()

```
void SetBlockSize(size_t blockSize)
virtual status_t SetSize(off_t numBytes)
```

SetBlockSize() sets the size of the memory blocks that the BMallocIO object deals with. The object allocates memory in multiples of the block size. The default is 256 bytes.

SetSize() sets the size of allocated memory to *numBytes* (modulo the block size). Shrinking the buffer should always be successful (B_OK); if the buffer can't be grown, B_NO_MEMORY is returned.

WriteAt()

```
virtual ssize_t WriteAt(off_t position, const void *buffer, size_t numBytes)
```

Copies *numBytes* bytes of data from *buffer* into the object's data beginning at *position*.

A successful WriteAt() always returns *numBytes*—WriteAt() reallocates the buffer (in multiples of the block size) if it needs more room. If the reallocation fails, this function returns B_NO_MEMORY.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMallocIO Function Summary

Function Summary

Function	Description
Buffer()	Returns a pointer to the allocated memory
BufferLength()	Returns the number of data bytes in the buffer
BMallocIO()	Constructor
~BMallocIO()	Destructor
Position()	Returns the position of the object's data pointer
ReadAt()	Copies data from the object into a buffer
Seek()	Moves the object's data pointer
SetBlockSize()	Sets the allocation block size
SetSize()	Sets the size of the allocated memory
WriteAt()	Writes data into the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BCheckBox

Derived from: public BControl
Declared in: be/interface/CheckBox.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BCheckBox object draws a labeled check box on-screen and responds to a keyboard action or a click by changing the state of the object: When the user "checks" the check box, the object displays an "x" within its border, its state is set to `B_CONTROL_ON`, and the invocation message is sent to the object's target. When the user "unchecks" the check box, the "x" is removed, the state is set to `B_CONTROL_OFF`, and the invocation message is sent.

Constructor and Destructor

BCheckBox()

```
BCheckBox(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BCheckBox(BMessage *archive)
```

Initializes the BCheckBox by passing all arguments to the BControl constructor (see `BControl::BControl()` for more information on the arguments.)

If necessary, the bottom coordinate of the object's frame rectangle is increased to accommodate the height of the label, given the object's font (`be_plain_font`, by default).

Freshly constructed, the object's state is `B_CONTROL_OFF`. Like all controls, the check box's view and low colors are set to those of its parent when the object is attached to a window.

~BCheckBox()

```
virtual ~BCheckBox()
```

Deletes the BCheckBox object.

Member Functions

All of BCheckBox's member functions are implementations of functions declared by its superclasses. For information on specific functions, look in BControl, then in BView, and so on up the inheritance hierarchy.

Archived Fields

See "Archived Fields" [BControl]. BCheckbox doesn't add any of its own fields to the archive message.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BCheckBox Function Summary

Function Summary

Function	Description
BCheckBox()	Constructor
~BCheckBox()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPictureButton

Derived from: BControl > BView > BHandler > BArchivable
Mix-in classes: BInvoker
Declared in: be/interface/PictureButton.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BPictureButton draws a button that displays an image (a BPicture object) rather than a text label. A BPictureButton can be either a one-state or a two-state control :

- **One-state**. Acts like a normal BButton: The value is `B_CONTROL_ON` while the object is being pressed and `B_CONTROL_OFF` otherwise. The images for these two states should be different; by convention, the "on" image is a highlighted version of the "off" image. If the button can be disabled, it needs an additional "disabled" image.
- **Two-state**. Acts like a checkbox: Its value switches between `B_CONTROL_ON` and `B_CONTROL_OFF` every time the user presses (and releases) the button. If the button can be disabled, it needs two additional "disabled" images: One for disabled-while-on and the other for disabled-while-off.

Constructor and Destructor

BPictureButton()

```
BPictureButton(BRect frame, const char* name,
    BPicture *off,
    BPicture *on,
    BMessage *message,
    uint32 behavior = B_ONE_STATE_BUTTON,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BPictureButton(BMessage *archive)
```

Creates a new BPictureButton. The *off* and an *on* images correspond to the object's **B_CONTROL_OFF** and **B_CONTROL_ON** values; the *behavior* argument sets the object to be a one-state (**B_ONE_STATE_BUTTON**) or two-state (**B_TWO_STATE_BUTTON**) control, as explained in the class description. The other arguments are inherited from the BView and BControl constructors. The object's initial value is **B_CONTROL_OFF**.

If the BPictureButton can be disabled, it needs additional "disabled" images, as set through **SetDisabledOff()** and **SetDisabledOn()**.

The BPictureButton copies all BPictures that are passed to it. It's the caller's responsibility to free the BPicture objects that are passed as arguments.

~BPictureButton()

```
virtual ~BPictureButton()
```

Deletes the object and its data.

Member Functions

- Behavior() see SetBehavior()
- DisabledOff(), DisabledOn(), EnabledOff(), EnabledOn() see SetEnabledOff()
- BView::Draw()
- KeyDown() (implementation detail) see BControl::KeyDown()
- BControl::MouseDown()

- SetBehavior() , Behavior()

virtual void **SetBehavior(**uint32 *behavior***)**
uint32 **Behavior(**void**)** const

These functions set and return whether the BPictureButton is a one-state (**B_ONE_STATE_BUTTON**) or a two-state (**B_TWO_STATE_BUTTON**) control. A one-state object acts like a normal button: It's on while the user is pressing it, and off otherwise. A two-state object switches to the opposite state each time the user presses and release the button.

- SetEnabledOff() , SetEnabledOn() , SetDisabledOff() , SetDisabledOn() , EnabledOff() , EnabledOn() , DisabledOff() , DisabledOn

virtual void **SetEnabledOff(**BPictureBControl::MouseDown() **picture***)**
BPicture ***EnabledOff(**void**)** const
virtual void **SetEnabledOn(**BPicture ***picture***)**
BPicture ***EnabledOn(**void**)** const
virtual void **SetDisabledOff(**BPicture ***picture***)**
BPicture ***DisabledOff(**void**)** const
virtual void **SetDisabledOn(**BPicture ***picture***)**
BPicture ***DisabledOn(**void**)** const

These pairs of functions set and return one of the four images the BPictureButton displays: enabled-and-on, enabled-and-off, disabled-and-on, and disabled-and-off, respectively. If this is a one-state object, the disabled-and-on image needn't be set since a disabled one-state control can never be on.

The BPicture-retrieving functions return **NULL** if the requested image hasn't been set.

The BPictureButton copies all BPictures that are passed to it. It's the caller's responsibility to free the BPicture objects that are passed as arguments.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_e_on" (deep copy only)	`B_MESSAGE_TYPE`	BPicture displayed when control is enabled and on.
"_e_off" (deep copy only)	`B_MESSAGE_TYPE`	BPicture displayed when control is enabled and off.
"_d_on" (deep copy only)	`B_MESSAGE_TYPE`	BPicture displayed when control is disabled and on.
"_d_off" (deep copy only)	`B_MESSAGE_TYPE`	BPicture displayed when control is disabled and off.
"_behave"	`B_INT32_TYPE`	BPictureButton behavior either **`B_ONE_STATE_BUTTON`** or **`B_TWO_STATE_BUTTON`**

See also: `BArchivable::Archive()`

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPictureButton Function Summary

Function Summary

Function	Description
Behavior()	Returns the object's behavior (one-state or two-state)
DisabledOff()	Sets the image that's displayed when the button is disabled and on
EnabledOff()	Sets the image that's displayed when the button is disabled and off
EnabledOn()	Returns the image that's displayed when the button is disabled and off
BPictureButton()	Constructor
~BPictureButton()	Destructor
SetBehavior()	Sets the object to be a one-state or a two-state object
SetDisabledOff()	Sets the image that's displayed when the button is enabled and on
SetDisabledOn()	Returns the image that's displayed when the button is enabled and on
SetEnabledOff()	Sets the image that's displayed when the button is enabled and off
SetEnabledOn()	Returns the image that's displayed when the button is enabled and off

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPicture

Derived from: none
Mix-in classes: BArchivable
Declared in: be/interface/Picture.h
Library: libbe.so
Summary: *more...*

A BPicture object represents a set of drawing instructions that are executed when the object is passed to BView's `DrawPicture()` function. Because it contains drawing instructions rather than an actual image, a BPicture (unlike a BBitmap) is independent of the resolution of the display device.

Recording a Picture

To start recording into a BPicture, you pass a BPicture object to **`BView::BeginPicture()`**. All drawing instructions that are executed by the BView are recorded into the BPicture object. When you're done recording, you call **`BView::EndPicture()`**, which passes back a pointer to the recorded object. For example:

```
BPicture *myPict;
someView->BeginPicture(new BPicture);
/* drawing code goes here*/
myPict = someView->EndPicture();
```

Only drawing that the BView does is recorded; drawing done by children and other views attached to the window is ignored, as is everything except drawing code.

Drawing instructions that are captured between **`BeginPicture()`** and **`EndPicture()`** are not renedered on-screen; ignored instructions may be rendered if they draw into the visible region of an on-screen window.

Any picture data in the BPicture passed to **`BeginPicture()`** is cleared; if you'd instead like to append to the BPicture, begin the picture recording with **`AppendPicture()`** instead. As with **`BeginPicture()`**, each **`AppendPicture()`** must have a corresponding **`EndPicture()`**.

The Picture Definition

The picture captures everything that affects the image that's drawn. It takes a snapshot of the BView's graphics state—the pen size, high and low colors, font size, and so on—when **`BeginPicture()`** is called. It then captures all subsequent modifications to those parameters, such as calls to **`MovePenTo()`**, **`SetLowColor()`**, and **`SetFontSize()`**. The recorded graphics state is used when the picture is drawn (through **`BView::DrawPicture()`**).

The picture records all primitive drawing instruction (**`DrawBitmap()`**, **`StrokeEllipse()`**, **`FillRect()`**, etc.) and will even record calls to **`DrawPicture()`**.

The picture makes its own copy of any data that's passed during the recording session, including bitmaps passed to **`DrawBitmap()`** and picture data passed to **`DrawPicture()`**.

Constructor and Destructor

BPicture()

```
BPicture(void)
BPicture(const BPicture &picture)
BPicture(BMessage *archive)
```

Initializes the BPicture object by ensuring that it's empty, or by copying data from another *picture* or *archive* of a BPicture object.

~BPicture()

```
virtual ~BPicture()
```

Destroys the Application Server's record of the BPicture object and deletes all its picture data.

Member Functions

Play()

virtual status_t ***Play**(void ***callBackTable*, int32 *tableEntries*, void **user*)**

Plays back a picture using a user's rendering functions. The functions are passed in *callBackTable*, an array of function pointers. *tableEntries* contains the number of functions in the table. The functions perform various tasks such as drawing lines and text. *user* is passed to each function, providing a hook for passing additional data to the call back functions. The functions, along with their positions in the call back table, are detailed below.

Index	Function prototype
0	no operation
1	MovePenBy(void *user, BPoint delta)
2	StrokeLine(void *user, BPoint start, BPoint end)
3	StrokeRect(void *user, BRect rect)
4	FillRect(void *user, BRect rect)
5	StrokeRoundRect(void *user, BRect rect, BPoint radii)
6	FillRoundRect(void *user, BRect rect, BPoint radii)
7	StrokeBezier(void *user, BPoint *control)
8	FillBezier(void *user, BPoint *control)
9	StrokeArc(void *user, BPoint center, BPoint radii, float startTheta, float arcTheta)
10	FillArc(void *user, BPoint center, BPoint radii, float startTheta, float arcTheta)
11	StrokeEllipse(void *user, BPoint center, BPoint radii)
12	FillEllipse(void *user, BPoint center, BPoint radii)
13	StrokePolygon(void *user, int32 numPoints, BPoint *points, bool isClosed)
14	FillPolygon(void *user, int32 numPoints, BPoint *points, bool isClosed)
15	Reserved
16	Reserved
17	DrawString(void *user, char *string, float deltax, float deltay)
18	DrawPixels(void *user, BRect src, BRect dest, int32 width, int32 height, int32 bytesPerRow, int32 pixelFormat, int32 flags, void *data)
19	Reserved
20	SetClippingRects(void *user, BRect *rects, uint32 numRects)
21	Reserved
22	PushState(void *user)
23	PopState(void *user)
24	EnterStateChange(void *user)

25	ExitStateChange(void *user)
26	EnterFontState(void *user)
27	ExitFontState(void *user)
28	SetOrigin(void *user, BPoint pt)
29	SetPenLocation(void *user, BPoint pt)
30	SetDrawingMode(void *user, drawing_mode mode)
31	SetLineMode(void *user, cap_mode capMode, join_mode joinMode, float miterLimit)
32	SetPenSize(void *user, float size)
33	SetForeColor(void *user, rgb_color color)
34	SetBackColor(void *user, rgb_color color)
35	SetStipplePattern(void *user, pattern p)
36	SetScale(void *user, float scale)
37	SetFontFamily(void *user, char *family)
38	SetFontStyle(void *user, char *style)
39	SetFontSpacing(void *user, int32 spacing)
40	SetFontSize(void *user, float size)
41	SetFontRotate(void *user, float rotation)
42	SetFontEncoding(void *user, int32 encoding)
43	SetFontFlags(void *user, int32 flags)
44	SetFontShear(void *user, float shear)
45	Reserved
46	SetFontFace(void *user, int32 flags)

While many of these functions are similar to those found in BView, there are some important differences:

- The return value of the functions is ignored.
- The Fill... and Stroke... functions do not explicitly specify patterns. Instead, they should be drawn in the current pattern, as set by the SetStipplePattern() (callback #35). Note that there is no equivalent to SetStipplePattern() in BView.
- The deltax and deltay arguments passed to DrawString() are escapement delta values; the string should be drawn at the current pen position.
- MovePenBy() uses a BPoint to specify the amount to move the pen. The x component of the BPoint gives the x offset and the y component the y offset.
- Similarly, ...RoundRect() and ...Ellipse() use a BPoint to specify the two separate radius components. The x component gives the x radius and the y component the y radius.
- DrawPixels() is a BPicture-specific primitive for rendering bitmaps. It is a request to copy the *src* rectangle from the raw color information in *data* to the *dest* rectangle of the

current rendering area. *width*, *height*, *bytesPerRow*, and *pixelFormat* provide all the information necessary to interpret data. *flags* is currently always zero and should be ignored. *src* and *dest* need not have the same dimensions; in these cases, the function should scale the bitmap appropriately.

- SetClippingRects() is a BPicture-specific primitive approximating **`ConstrainClippingRegion()`**. It instructs the renderer to replace the current clipping region with the union of the rectangles passed to SetClippingRects().
- Changes in the graphics state are sandwiched between calls to EnterStateChange() and ExitStateChange(). State change functions will only be called between these functions. No other call backs will be called between these functions. State change functions are all Set... functions in addition to EnterFontState() and ExitFontState().
- Similarly, changes to the font state are sandwiched between calls to EnterFontState() and ExitFontState(). Font state change functions will only be called between these functions. No other call backs will be called between these functions. Font state change functions are all SetFont... functions. Many of the font state functions are found in BFont rather than BView.
- SetFontRotate() sets the rotation of the font. Unlike the **`BFont::SetRotation()`** function, the angle here is specified in radians, rather than in degrees. You can convert the value into degrees by using the forumla:

degrees = (rotation*180.0)/3.14159265369);

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_ver"	`B_INT32_TYPE`	Always 1.
"_endian"	`B_INT8_TYPE`	Endianness of the data. Always `B_HOST_IS_BENDIAN`.
"_data"	`B_RAW_TYPE`	The BPicture data.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPicture Function Summary

Function Summary

Function	Description
BPicture()	Constructor
~BPicture()	Destructor
Play()	Renders a BPicture using user-supplied functions

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global Functions, Constants, and Defined Types

Library: libgame.so
more...

The Game Kit has just one global function, **`set_mouse_position()`**. It should be called only by an application that has control of the screen through a BWindowScreen object. Otherwise, the kit borrows constants and types defined for graphics card drivers and by other parts of the BeOS. Those that are not documented for the Interface and Support Kits are listed here.

Global Functions

set_mouse_position()

Declared in: be/game/WindowScreen.h

```
void set_mouse_position(int32 x, int32 y)
```

Moves the cursor hot spot to (*x*, *y*) in the screen coordinate system, where *x* is a left-to-right index to a pixel column and *y* is a top-to-bottom index to a pixel row. The origin of the coordinate system is the left top pixel of the display area of the main screen.

This function should be called only while the application has a direct connection to the frame buffer through a BWindowScreen object.

Constants

Control Flags

Declared in: be/add-ons/graphics/GraphicsCard.h

B_CRT_CONTROL
B_GAMMA_CONTROL
B_FRAME_BUFFER_CONTROL

These flags report the driver's ability to control the CRT display, make gamma corrections, and permit nonstandard configurations of the frame buffer. Only the last has any meaning for the Game Kit.

See also: `CardInfo()`

gs_attributes

Declared in: be/game/GameSoundDefs.h

Constant	Meaning
B_GS_MAIN_GAIN	Main gain control, in decibels. The main gain doesn't support ramping.
B_GS_CD_THROUGH_GAIN	Gain on the CD through, in decibels.
B_GS_GAIN	Gain on the sound, in decibels.
B_GS_PAN	Pan position of the sound. -1.0 for far left, 0 for middle, 1.0 for far right.
B_GS_SAMPLING_RATE	Sampling rate in hertz.
B_GS_LOOPING	If the attribute's value is nonzero, the sound automatically loops. If it's 0, the sound plays through just once.
B_GS_FIRST_PRIVATE_ATTRIBUTE	Beginning of private attribute range
B_GS_FIRST_USER_ATTRIBUTE	Beginning of user attribute range.

These are the various possible game sound attributes. The range between `B_GS_FIRST_PRIVATE_ATTRIBUTE` and `B_GS_FIRST_USER_ATTRIBUTE` are reserved; if you need custom attributes, use values `B_GS_FIRST_USER_ATTRIBUTE` and higher.

Defined Types

frame_buffer_info

Declared in: be/add-ons/graphics/GraphicsCard.h

```
typedef struct {
        short bits_per_pixel;
        short bytes_per_row;
        short width;
        short height;
        short display_width;
        short display_height;
        short display_x;
        short display_y;
      } frame_buffer_info
```

This structure is used to report the current configuration of the frame buffer.

See also: `FrameBufferInfo()`

graphics_card_hook

Declared in: be/add-ons/graphics/GraphicsCard.h

```
typedef void (*graphics_card_hook)(void)
```

This is the general type declaration for a graphics card hook function. Specific hook functions will in fact declare various sets of arguments and all return a `status_t` error code rather than `void`.

See also: `CardHookAt()`, "Graphics Card Hook Functions" on page18

graphics_card_info

Declared in: be/add-ons/graphics/GraphicsCard.h

```
typedef struct {
        short version;
        short id;
        void *frame_buffer;
        char rgba_order[4];
        short flags;
        short bits_per_pixel;
        short bytes_per_row;
        short width;
```

```
    short height;
} graphics_card_info
```

Drivers use this structure to supply information about themselves and the current configuration of the frame buffer to the Application Server and the BWindowScreen class.

See also: `CardInfo()`

---

gs_attribute

Declared in: be/game/GameSoundDefs.h

```
struct gs_attribute {
    int32 attribute;
    bigtime_t duration;
    float value;
    uint32 flags;
};
```

Defines an attribute. An attribute consists of an `attribute` number from `gs_attributes`, a `duration` indicating the period of time, in microseconds, over which the attribute's change takes effect, and a target `value`. Additional `flags` can be specified for the attribute as well; these vary depending on the attribute.

Currently, there are no flags defined for any of the predefined attributes.

---

gs_attribute_info

Declared in: be/game/GameSoundDefs.h

```
struct gs_attribute_info {
    int32 attribute;
    float granularity;
    float minimum;
    float maximum;
};
```

Describes the possible values the `attribute` can take. The `granularity` field indicates how finely the value of the attribute can be controlled, and `minimum` and `maximum` specify the minimum and maximum values the attribute can take on.

---

gs_audio_format

Declared in: be/game/GameSoundDefs.h

```
struct gs_audio_format {
    enum format {
      B_GS_U8 = 0x11,
```

```
    B_GS_S16 = 0x2,
    B_GS_F = 0x24,
    B_GS_S32 = 0x4
  };
  float frame_rate;
  uint32 channel_count;
  uint32 format;
  uint32 byte_order;
  size_t buffer_size;
};
```

This structure describes the format of a game sound. The format enum lists the possible sound sample formats supported by the Game Kit:

Constant	Meaning
B_GS_U8	Unsigned 8-bit integer.
B_GS_S16	Signed 16-bit integer.
B_GS_F	Floating-point.
B_GS_S32	Signed 32-bit integer.

- The **frame_rate** indicates how many frames per second of audio should be played.
- The **channel_count** field indicates the number of audio channels the sound uses.
- The **format** field, which specifies the format of the audio data to be played must be one of the values declared in the **format** enum.
- The **byte_order** field specifies the byte order of the sound to be played.
- The **buffer_size** is used to specify how large the audio buffers used to play the sound should be. You can specify zero if you want the Game Kit to determine an appropriate size for you.

---

gs_id

Declared in: be/game/GameSoundDefs.h

```
typedef int32 gs_id;
```

This type is used for game sound ID numbers.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BGameSound

Derived from: none
Declared in: be/game/GameSound.h
Library: libgame.so
Allocation: Constructor only
Summary: *more...*

The BGameSound class is the base class for other game sound classes. You never use this class directly; instead, you should use the derived classes.

The Sound of Explosions in the Morning

Nothing makes the player happy like the sound of lots of explosions, weird alien things, and other thrilling audio pleasures. The various BGameSound-derived classes make this a snap.

Choosing a Player Class

Playing a sound is as simple as instantiating an object derived from the most appropriate class and calling **`StartPlaying()`** on it. How do you determine the most appropriate class? Here are some hints:

- If you have a sound effect that's short and is likely to be played often, or needs to be played with minimal latency, use the BSimpleGameSound class. An arcade game's explosions definitely fit into this class.
- If you have a sound effect that's very long, or doesn't need to be played very often, or whose latency doesn't matter much, use the BFileGameSound class. Music loops can take advantage of this class.
- If you want to be able to push audio buffers at the system, instead of having a callback that fills buffers, use the BPushGameSound class.

Most games' needs will be filled by the BSimpleGameSound and BFileGameSound classes. And, of course, you can write your own derived classes if none of these classes meet your requirements.

Polyphony

Sometimes the player needs to hear multiple copies of the same sound at the same time; for instance, if they shoot a three-way splitting weapon and hit two targets, they need to hear two explosions at once. Each BGameSound (or BGameSound-derived) object can play only once at a time, so you'll need to use cloned copies of the sound, one for each channel of polyphony you allow.

This can be done with ease using a simple class like this:

```
class MultipleEffect {
public:
   MultipleEffect(BGameSound * sound, int polyphony) {
      m_fx = new BGameSound *[polyphony];
      m_fx[0] = sound;
      for (int ix=1; ix<polyphony; ix++) {
         m_fx[ix] = sound->Clone();
      }
      m_current = 0;
      m_polyphony = polyphony;
   }
   void StartPlaying() {
```

```
      int id = atomic_add(&m_current, 1) % m_polyphony;
      m_fx[id]->StartPlaying();
   }

private:
   int32 m_current;
   int32 m_polyphony;
   BGameSound *m_fx;
};
```

This class' constructor lets you specify a BGameSound-derived sound and the maximum number of times it can be playing at once. It creates the appropriate number of clones, and its `StartPlaying()` implementation automatically selects the oldest one and reuses it. Since `StartPlaying()` restarts a sound from the beginning if it's called on a playing sound, this works out well—if polyphony is 3, and there are already three sounds playing, the oldest one will be reset and played from the beginning.

In the current version of the BeOS, when you clone a BSimpleGameSound, the sound data buffer is also cloned, so you'll have multiple copies of the sound effect in memory. Keep this in mind as you write your code, as you can rapidly use a lot of memory this way.

Performance Verification

There are a wide variety of performance concerns, and of course you want your game to work on as many systems as possible. Be recommends that you test your application in at least these three situations:

- Real-time audio disabled, using an ISA sound card (such as an AWE64 or equivalent card).
- Real-time audio enabled, using a PCI sound card (SoundBlaster PCI 64 or equivalent).
- Real-time audio enabled, using an E-mu 10k sound card. This sound card has its own node, so its behavior is a little different, and is therefore worth special testing attention.

This isn't intended to be an exhaustive list; you should always test your software in as many configurations as possible.

Constructor and Destructor

BGameSound()

```
BGameSound(BGameSoundDevice *device = NULL)
```

Initializes the sound object.

Currently, *device* must always be **NULL**.

~BGameSound

```
virtual ~BGameSound()
```

Releases any memory used by the BGameSound.

If your node has created and set BBufferGroups for any producers, you should delete them in the destructor.

Member Functions

Clone()

```
virtual BGameSound *Clone(void) const = 0
```

Returns a copy of the game sound object. Not implemented in this base class; derived classes must implement it.

Device()

```
BGameSoundDevice *Device(void) const
```

Returns a pointer to the BGameSoundDevice responsible for playing the sound. This is **NULL** if the default device is being used.

Format()

```
const gs_audio_format &Format(void) const
```

Returns a **gs_audio_format** structure describing the format of the sound.

Gain() see SetGain()

GetAttributes() see SetAttributes()

ID()

```
gs_id ID(void) const
```

Returns the ID of the sound attached to the BGameSound object. This is 0 if no sound has been selected.

Init()

protected:

```
status_t Init(gs_id soundHandle)
```

Specifies the sound to be played by the object.

RETURN CODES

- B_OK. The initialization was successful.
- **B_ERROR**. The constructor failed.
- Other errors. The player may return other errors.

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t indicating whether or not the object was successfully initialized. A return value of **B_OK** means everything's fine; any other value means an error occurred in the constructor.

IsPlaying()

```
virtual bool IsPlaying(void)
```

Returns **true** if the sound is currently playing; otherwise this function returns **false**.

LockMemoryPool()

```
static status_t LockMemoryPool(bool lockInCore)
```

This function lets you specify whether or not the BGameSound's memory pool (from which buffers are drawn) is locked in memory. Locking the pool may increase performance, but will increase your memory requirements and may reduce the ability of the computer to use virtual memory efficiently.

You can only set the lock state from within the constructor of a derived class. Once the state has been set, it can't be changed.

RETURN CODES

- B_OK. The pool's lock state has been set.
- **EALREADY**. The lock state has already been set.

Pan() see SetPan()

SetAttributes() , GetAttributes()

virtual status_t **SetAttributes(**gs_attribute *_inAttributes_, size_t _inAttributeCount_**)**
virtual status_t **GetAttributes(**gs_attribute *_ioAttributes_, size_t _inAttributeCount_**)**

`SetAttributes()` sets the sound's attributes to match those in the *inAttributes* array; *inAttributeCount* indicates how many attributes are in the list.

`GetAttributes()` returns the sound's current attributes. The *ioAttributes* list contains a list of *inAttributeCount* **`gs_attribute`** structures; each entry indicates in its `attribute` field which attribute is to be stuffed into that slot in the array. On return, these attributes are filled out.

RETURN CODES

- B_OK. The attribute changes were recorded.
- **`B_ERROR`**. The player wasn't initialized properly.

SetGain() , Gain()

status_t **SetGain(**float *gain*, bigtime_t *rampDuration* = 0**)**
float **Gain(**void**)**

`SetGain()` sets the sound's gain (volume). If *rampDuration* is nonzero, the change in gain will take *rampDuration* to occur—the gain will fade between the current value and the new value over the specified number of microseconds. If *rampDuration* is 0, the change will be immediate.

`Gain()` returns the sound's current gain.

The gain ranges from 0.0 (silent) to 1.0 (maximum gain).

RETURN CODES

- B_OK. The gain change was recorded.
- **`B_ERROR`**. The object hasn't been initialized properly.
- Other errors. The derived class may return other appropriate errors.

SetInitError()

protected:

status_t **SetInitError(**status_t *initError***)**

Sets the error code that **`InitCheck()`** will return. Derived classes should call this from their constructors.

SetMaxSoundCount()

static int32 **SetMaxSoundCount(**int32 *maxSoundCount***)**

This function lets you specify the maximum number of BGameSound-derived objects that can be in use at once. This value must be between 32 and 1024; if it's outside this range, it gets pinned to 32 or 1024. The actual value set is returned.

SetMemoryPoolSize()

status_t **SetMemoryPoolSize(**size_t *poolSize***)**

Sets the size of the memory pool from which sound buffers can be obtained to the number of bytes specified by *poolSize*. This value must be between 128kB and 4MB, or **B_BAD_VALUE** will be returned.

You can't change the size of the memory pool once it's been allocated.

In general you'll only call this function if you're implementing a new BGameSound subclass.

RETURN CODES

- B_OK. The memory pool size has been set.
- **B_BAD_VALUE**. The requested pool size is invalid.
- **EALREADY**. The memory pool has already been allocated; you can't change its size.

SetPan() , Pan()

status_t **SetPan(**float *pan*, bigtime_t *rampDuration* = 0**)**

float **Pan(**void**)**

`SetPan()` sets the sound's pan. This value, which ranges from -1.0 to 1.0, indicates the apparent position of the sound (from absolute left to absolute right). A value of 0 indicates that the sound should appear to come from directly in front of the user. If *rampDuration* is nonzero, the change in position will take *rampDuration* to occur—the sound will seem to move from the initial position to the new position over the specified number of microseconds. If *rampDuration* is 0, the change will be immediate.

`Pan()` returns the sound's current pan.

RETURN CODES

- B_OK. The pan change was recorded.

- **B_ERROR**. The object hasn't been initialized properly.
- Other errors. The derived class may return other appropriate errors.

StartPlaying() , StopPlaying()

virtual status_t **StartPlaying**(void)
virtual status_t **StopPlaying**(void)

`StartPlaying()` begins playback of the sound. `StopPlaying()` ends playback.

If you call **`StartPlaying()`** on a sound that's already playing, it stops and restarts from the beginning.

These return **B_OK** if all is well, or an appropriate error code if not; the error depends on the derived class being used.

RETURN CODES

- B_OK. No error.
- **EALREADY**. The sound is already playing (if you called `StartPlaying()`) or not playing (if you called `StopPlaying()`).
- Other errors, as determined by the subclass.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BGameSound Function Summary

Function Summary

Function	Description
Clone()	Returns a clone of the sound
Device()	Returns a pointer to the BGameSoundDevice assigned to play the sound
Format()	Returns the audio format of the sound
Gain()	Returns the sound's current gain
BGameSound()	Constructor
GetAttributes()	Returns the sound's attributes
ID()	Returns the ID of the sound attached to the BGameSound
Init()	Sets the sound to be played by the object
InitCheck()	Indicates whether or not the object was initialized successfully
IsPlaying()	Indicates whether or not the sound is playing
LockMemoryPool()	Locks or unlocks the BGameSound's memory pool
Pan()	Returns the sound's current left-right placement
SetAttributes()	Sets the sound's attributes
SetGain()	Sets the sound's gain
SetInitError()	Sets the object's initialization error

SetMaxSoundCount()	Specifies the maximum number of sounds that can be in use at once
SetMemoryPoolSize()	Sets the size, in bytes, of the sound's memory pool
SetPan()	Sets the sound's left-right placement
StartPlaying()	Starts playing the sound
StopPlaying()	Stops the sound playback

The Be Book **The Game Kit** **The Game Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSimpleGameSound

Derived from: BGameSound
Declared in: be/game/SimpleGameSound.h
Library: libgame.so
Allocation: Constructor only
Summary: *more...*

The BSimpleGameSound class represents simple sound effects that don't change, and remain in memory.

Using BSimpleGameSound is, well, simple:

```
BSimpleGameSound *mysound = new BSimpleGameSound("soundfile.wav");
...
mysound.StartPlaying();
```

This snippet uses BSimpleGameSound to create an object that can be used to play the sound effect in the file "soundfile.wav". Playing the sound is as simple as calling **StartPlaying()**.

In the current version of the BeOS, when you clone a BSimpleGameSound, the sound data buffer is also cloned, so you'll have multiple copies of the sound effect in memory. Keep this in mind as you write your code, as you can rapidly use a lot of memory this way.

Constructor and Destructor

BSimpleGameSound()

```
BSimpleGameSound(const entry_ref *inFile,
    BGameSoundDevice *device = NULL)
BSimpleGameSound(const char *inFile,
    BGameSoundDevice *device = NULL)
BSimpleGameSound(const void *inData, size_t inFrameCount,
    const gs_audio_format *format,
    BGameSoundDevice *device = NULL)
BSimpleGameSound(const BSimpleGameSound &other)
```

Prepares the object to play the specified sound. The first form of the constructor preloads the entire sound specified by *inFile* into memory, while the second accepts a *inFile* as a pathname string.

The third form takes *inData* as a pointer to sound data already in memory; this sound data is copied into a buffer owned by the BSimpleGameSound object; once the constructor returns, you can delete the original data if you wish. The data is copied since some sound cards have an onboard sound buffer, and the API allows support for these devices (whether or not drivers exist that do this is another story).

inFrameCount indicates how many frames of audio are in that buffer, and *format* indicates the format of the audio data. The size of the Media Kit buffers used by the object is determined by the gs_audio_format structure's **`buffer_size`** field. If the value this field is zero, the Game Kit determines an appropriate buffer size for you.

In both cases, *device* specifies the sound device that should be used for playing the sound; **`NULL`** uses the default sound player.

Currently, *device* must always be **`NULL`**.

The final form of the constructor duplicates another BSimpleGameSound object.

After instantiating the BSimpleGameSound object, you should call **`InitCheck()`** to determine whether or not the sound was successfully created.

~BSimpleGameSound

```
virtual ~BSimpleGameSound()
```

A typical destructor.

Member Functions

InitCheck()

status_t **InitCheck(**void**)** const

Returns a status_t indicating whether or not the object was successfully instantiated.

RETURN CODES

- B_OK. The sound was successfully initialized.
- **B_ERROR**. Unable to create a sound player.
- **B_NO_MEMORY**. Can't get enough memory to preload the sound.
- *Other errors*. The sound player may return errors.

SetIsLooping() , IsLooping()

status_t **SetIsLooping(**bool *looping***)**
bool **IsLooping(**void**)** const

SetIsLooping() turns looping of the sound on if *looping* is **true**, or off if *looping* is **false**.

`IsLooping()` returns a flag indicating whether or not looping is currently enabled.

RETURN CODES

- B_OK. Looping was turned on or off without error.
- **B_ERROR**. The player wasn't initialized properly.
-

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSimpleGameSound Function Summary

Function Summary

Function	Description
InitCheck()	Indicates whether or not the object was initialized successfully
IsLooping()	Indicates whether or not looping is enabled
SetIsLooping()	Turns looping of the sound on or off
BSimpleGameSound()	Constructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFileGameSound

Derived from: BStreamingGameSound
Declared in: be/game/FileGameSound.h
Library: libgame.so
Allocation: Constructor only
Summary: *more...*

If you want to play back audio from a disk file, such as background music, or low-priority sound effects, BFileGameSound is for you. Keep in mind that if the sound needs to play at precise moments, or latency is an issue, that BFileGameSound may not be appropriate for your needs.

Using BFileGameSound is easy, and it supports automatically looping sounds. The following code snippet starts up the theme music for the hot new game "Mystery Warriors From the Doomed Planet Z":

```
BFileGameSound themeSong("music/theme.aif", true);
themeSong.StartPlaying();
```

This starts up a looped sound from the file **music/theme.aif**. If you want the theme to only play through once, just specify **`false`** instead of **`true`** for the looping argument to the BFileGameSound constructor.

You can pause the BFileGameSound by calling **`SetPaused()`**:

```
themeSong.SetPaused(true, 0);
```

This pauses the sound, effective immediately. The pause can optionally be ramped, so that the sound slows down or speeds up to reach the new setting. For example:

```
themeSong.SetPaused(true, 2);
```

This causes the sound to slow down over the course of two seconds, until it's stopped.

The inverse is also possible:

```
themeSong.SetPaused(false, 0);
```

This resumes playback immediately. You can ramp the resume by specifying a non-zero value for the ramp time.

BFileGameSound uses the BMediaFile class to access the sound file. If BMediaFile can't identify the sound, the file is assumed to contain raw 44.1kHz, 16-bit stereo.

Constructor and Destructor

BFileGameSound()

```
BFileGameSound(const entry_ref *inFile,
    bool looping = true,
    BGameSoundDevice *device = NULL)
BFileGameSound(const char *inFile,
    bool looping = true,
    BGameSoundDevice *device = NULL)
```

Prepares the object to play the specified sound file, which can be specified either by entry_ref or pathname in the *inFile* argument.

If the looping flag is **true** (which it is by default), the sound automatically loops back to the beginning and replays when the end of the sound is reached. This is useful for easily playing background music (for example).

In both cases, *device* specifies the sound device that should be used for playing the sound; **NULL** uses the default sound player.

Currently, *device* must always be **NULL**.

After instantiating the BFileGameSound object, you should call **InitCheck()** to determine whether or not the sound was successfully created.

~BFileGameSound

```
virtual ~BFileGameSound()
```

A typical destructor.

Member Functions

InitCheck()

status_t **InitCheck**(void) const

Returns a status_t indicating whether or not the object was successfully instantiated.

RETURN CODES

- B_OK. The sound was successfully initialized.
- **B_ERROR**. Unable to create a sound player.
- **B_NO_MEMORY**. Can't get enough memory to preload the sound.
- *Other errors*. The sound player may return errors.

IsPaused() see SetPaused()

Preload()

status_t **Preload**(void)

`Preload()` preloads enough of the sound file into memory that starting playback of the sound won't cuase a delay while the first chunk of data is fetched from disk.

RETURN CODES

- B_OK. Preloading was successful.
- Port errors. Unable to communicate with the streaming sound port.

SetPaused() , IsPaused()

status_t **SetPaused**(bool *isPaused*, bigtime_t *rampTime*)
int32 **IsPaused**(void)

`SetPaused()` pauses if *isPaused* is **true**, and unpauses if *isPaused* is **false**. If *rampTime* is nonzero, the sound ramps up to speed (or down to stopped) instead of instantly changing state. The number of microseconds specified by *rampTime* indicates how long the change should take to complete.

`IsPaused()` returns value indicating whether or not the sound is paused, or if a pause is in process of being initiated (if ramping is underway). The result is one of these values:

- **`B_NOT_PAUSED`**—the sound is playing normally.
- **`B_PAUSE_IN_PROGRESS`**—the sound is ramping toward or away from a paused state.
- **`B_PAUSED`**—the sound is paused.

Because these constants are members of the BFileGameSound class, be sure to refer to them as **`BFileGameSound::B_NOT_PAUSED`** and so forth.

RETURN CODES

- B_OK. The pause occurred without error.
- **`EALREADY`**. The sound is already in the requested state.
-

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStreamingGameSound

Derived from: BGameSound
Declared in: be/game/StreamingGameSound.h
Library: libgame.so
Allocation: Constructor only
Summary: *more...*

The BStreamingGameSound class provides the ability to stream audio data. A hook function is called whenever a buffer needs to be filled with audio data. This hook function is set by calling `SetStreamHook()`.

Using this class requires special magic powers; unless you're directly in contact with Be about it, don't use it. You'll regret it later.

Constructor and Destructor

BStreamingGameSound()

```
BStreamingGameSound(size_t inBufferFrameCount,
    const gs_audio_format *format,
    size_t inBufferCount = 2,
    BGameSoundDevice *device = NULL)
```

protected:

```
BStreamingGameSound(BGameSoundDevice device)
```

Prepares the object to play streamed audio. *inBufferFrameCount* specifies the number of frames each audio buffer should be able to hold. *format* indicates the audio format that will be streamed. *inBufferCount* specifies the number of buffers to use, and, as always, *device* is the sound device to use for playback.

Currently, *device* must always be **NULL** to indicate that the default playback device should be used.

By default, two audio buffers are used.

~BStreamingGameSound

```
virtual ~BStreamingGameSound()
```

Stops playing the sound.

Member Functions

FillBuffer()

virtual void **FillBuffer(**void **inBuffer*, size_t *byteCount***)**

Fills the buffer specified by inBuffer with byteCount bytes of audio data.

In the BStreamingGameSound implemenation, this function calls the stream hook, if one exists.

See also: `SetStreamHook()`

InitCheck()

status_t **InitCheck(**void**)** const

Returns a status_t indicating whether or not the object was successfully instantiated.

RETURN CODES

- B_OK. The sound was successfully initialized.
- **`B_ERROR`**. Unable to create a sound player.
- Other errors. The sound player may return errors.

See also: the BStreamingGameSound constructor

Lock() , Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

`Lock()` locks the BStreamingGameSound to prevent unpleasant collisions in the land of multithreadedness; it returns **`true`** if it was able to lock the object, otherwise **`false`** is returned. `Unlock()` releases the lock.

SetAttributes()

virtual status_t **SetAttributes(**gs_attribute **inAttributes*, size_t *inAttributeCount***)**

`SetAttributes()` is implemented to disallow the **`B_GS_LOOPING`** attribute, since streamed sounds can't loop.

See also: **`BGameSound::SetAttributes()`**

SetParameters()

protected:

```
virtual status_t SetParameters(size_t inBufferFrameCount,
    const gs_audio_format *format,
    size_t inBufferCount)
```

Changes the BStreamingGameSound object's parameters. This lets you change the buffer size, audio format, and number of buffers after the object has been instantiated.

RETURN CODES

- B_OK. No error.
- **`B_ERROR`**. Couldn't change the parameters.

SetStreamHook()

```
virtual status_t SetStreamHook(void (*hook)(void *cookie, void *inBuffer,
    size_t byteCount, BStreamingGameSound *object),
        void *cookie)
```

Specfies the hook function to be called to fill buffers with audio data. *cookie* indicates the cookie pointer that will be passed to the hook function.

The inputs to the hook function are:

- *cookie*
  A user-defined value.
- *inBuffer*
  A pointer to the buffer to be filled with audio data.
- *byteCount*
  The size of the buffer in bytes.
- *object*
  A pointer to the BStreamingGameSound object.

RETURN CODES

- B_OK. The hook function was set without error.

- Unlock() see Lock()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

- BStreamingGameSound

Function Summary

Function Summary

Function	Description
FillBuffer()	Fills the specified buffer with the indicated number of bytes of sound data
InitCheck()	Indicates whether or not the object was initialized successfully
Lock()	Locks the sound for thread protection
SetAttributes()	Implementation detail
SetParameters()	Changes the streaming sound's parameters
SetStreamHook()	Sets the hook function that will be called to fill buffers with sound data
BStreamingGameSound()	Constructor
Unlock()	Unlocks the sound

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFileGameSound Function Summary

Function Summary

Function	Description
BFileGameSound()	Constructor
InitCheck()	Indicates whether or not the object was initialized successfully
IsPaused()	Indicates whether or not playback is paused
Preload()	Preloads sound from the file to make starting playback faster
SetPaused()	Pauses or unpauses the sound

The Be Book **The Game Kit** **The Game Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Media Kit

BBufferConsumer

Declared in: be/media/BufferConsumer.h
more...

BBufferConsumer()

Declared in: be/protected:

```
explicit BBufferConsumer(media_type consumer)
```

The *consumer* argument used to take a default value (**B_MEDIA_UNKNOWN_TYPE**); it no longer does.

BBufferProducer

Declared in: be/media/BufferProducer.h

BBufferProducer()

Declared in: be/protected:

```
explicit BBufferProducer(media_type producer)
```

The *producer* argument used to take a default value (**B_MEDIA_UNKNOWN_TYPE**); it no longer does.

---

suggestion_quality

```
enum suggestion_quality(
        B_ANY_QUALITY = 0,
    B_LOW_QUALITY  = 10,
    B_MEDIUM_QUALITY = 50,
    B_HIGH_QUALITY = 100)
```

These constants can be used as values for the *quality* argument to FormatSuggestionRequested(). Although you don't have to use these constants, it's expected that you'll restrict your request to 0-100.

BMediaAddOn

Declared in: be/media/MediaAddOn.h

The following functions are no longer pure virtual, they are just virtual: **InitCheck()**, **CountFlavors()**, **GetFlavorAt()**, **InstantiateNodeFor()**, **GetConfigurationFor()**, **WantsToStart()**, **AutoStart()**

NotifyFlavorChange()

Calling this will cause everyone concerned to be notified of the change to your flavor, and also cause the media server to re-scan your flavor info. It is thread safe. Note that this function is defined protected.

status_t NotifyFlavorChange()

BMediaFormats

Declared in: be/media/MediaFormats.h

MakeFormatFor()

```
status_t MakeFormatFor(const media_format_description * descs, int32 desc_count, media_format
io_format, uint32 flags = 0, void * _reserved = 0);
```

GetBeOSFormatFor()

```
static status_t GetBeOSFormatFor(uint32 fourcc, media_format * out_format, media_type type =
B_MEDIA_UNKNOWN_TYPE)
```

GetAVIFormatFor()

```
static status_t GetAVIFormatFor(uint32 fourcc, media_format * out_format, media_type type =
B_MEDIA_UNKNOWN_TYPE)
```

GetQuicktimeFormatFor

```
static status_t GetQuicktimeFormatFor(uint32 vendor, uint32 fourcc, media_format *out_format, m
type = B_MEDIA_UNKNOWN_TYPE)
```

get_next_encoder()

```
status_t get_next_encoder(int32 *cookie, media_codec_info *ei);
status_t get_next_encoder( int32 *cookie, const media_file_format *mfi, const media_format
*input_format, const media_format *output_format, media_codec_info *ei, media_format
*accepted_input_format, media_format *accepted_output_format)
```

- *mfi* comes from `get_next_file_format()`, pass `NULL` if you don't care.
- *input_format* is the type of data given to the encoder, wildcards are allowed.
- *output_format*: the desired output format, wildcards allowed.
- *ei*: information about the encoder.

- *accepted_input_format*: Type of data that the encoder will accept as input. Wildcards given in *input_format* will be specialized here.
- *accepted_output_format*: The type of data that the encoder will output. Wildcards in *output_format* will be specialized here.

---

does_file_accept_format()

```
bool does_file_accept_format(const media_file_format *mfi, const media_format *format);
```

---

typedef , media_avr_description

Supports the avr format

```
typedef struct {
    uint32 id;
} media_avr_description;
```

---

enum , make_format_flags

```
enum make_format_flags {
B_EXCLUSIVE = 0x1, //Fail if this format has already been registered
B_NO_MERGE = 0x2, //Don't re-number any formats if there are multiple clashing previous registrati
fail instead
B_SET_DEFAULT = 0x4 //Set the first format to be the default for the format family (when registerin
than one in the same family). Only use in Encoder add-ons.
}
```

BMediaEventLooper

Declared in: be/media/MediaEventLooper.h

ControlThread()

`thread_id` **`ControlThread`** ()

Protected function to return the id of the thread running this BMediaEventLooper.

BMediaFile

Declared in: be/media/MediaFile.h

BMediaFile()

Two new constructors for read-only access to files:

```
BMediaFile(const entry_ref * ref, int32 flags);
BMediaFile(BDataIO * source, int32 flags);
```

BMediaRoster

Declared in: be/media/mediaRoster.h

GetAudioOutput() , SetAudioOuput()

New versions of these functions.

```
status_t GetAudioOutput(media_node * out_node, int32 * out_input_id, BString * out_input_name)
status_t SetAudioOuput(const media_input & input_to_output)
```

GetAllInputsFor() , GetAllOutputFor()

```
status_t GetAllInputsFor(const media_node & node, media_input * out_inputs, int32 buf_num_inpu
* out_total_count)
status_t GetAllOutputsFor(const media_node & node, media_output * out_outputs, int32 buf_num_(
int32 * out_total_count)
```

GetFormatFor()

```
status_t GetFormatFor(const media_output & output, media_format * io_format, uint32 flags = 0)
status_t GetFormatFor(const media_input & input, media_format * io_format, uint32 flags = 0)
status_t GetFormatFor(const media_node & node, media_format * io_format, float quality =
B_MEDIA_ANY_QUALITY)
```

NodeIDFor()

```
media_node_id NodeIDFor(port_id source_or_destination_port)
```

Use to find the media node associated with a given port.

GetInstancesFor()

```
status_t GetInstancesFor(media_addon_id addon, int32 flavor, media_node_id * out_id, int32 * io_c
```

BMediaTrack

Declared in: be/media/MediaTrack.h

- `media_seek_type` enum has been extended with `B_MEDIA_SEEK_DIRECTION_MASK`.

ReadFrames()

New version of ReadFrames()

```
status_t ReadFrames(void *out_buffer, int64 *out_frameCount, media_header *mh, media_decode_inf
```

FindKeyFrameForTime() , FindKeyFrameForFrame()

```
status_t FindKeyFrameForTime(bigtime_t *inout_time, int32 flags=0) const;
status_t FindKeyFrameForFrame(int64 *inout_frame, int32 flags=0) const;
```

WriteFrames()

```
status_t WriteFrames(const void *data, int64 num_frames, media_encode_info *info);
status_t WriteFrames(const void *data, int64 num_frames, media_encode_info *info);
```

GetEncodeParameters() , GetEncodeParameters()

```
status_t GetEncodeParameters(encode_parameters *parameters) const;
status_t SetEncodeParameters(encode_parameters *parameters);
```

BSoundPlayer

Declared in: be/media/SoundPlayer.h

BSoundPlayer()

```
BSoundPlayer(const media_node & toNode, const media_multi_audio_format * format = NULL, const
name = NULL, const media_input * input = NULL, void (*PlayBuffer)(void *, void * buffer, size_t siz
media_raw_audio_format & format) = NULL, void (*Notifier)(void *, sound_player_notification wh
NULL, void * cookie = NULL)
```

New constructor to support the new `media_multi_audio_format` type.

GetVolumeInfo()

```
status_t GetVolumeInfo(media_node * out_node, int32 * out_parameter, float * out_min_dB, floa
out_max_dB)
```

SetVolumeDB() , VolumeDB()

```
float VolumeDB(bool forcePoll = false)
```

```
void SetVolumeDB(float volume_dB)
```

BTimedEventQueue

Declared in: be/media/TimedEventQueue.h

```
int32 EventCount() const
```

How many events in the queue?

Media Kit Miscellanea

suggestion_quality

Declared in: be/media/BufferProducer.h

```
enum suggestion_quality {
  B_ANY_QUALITY = 0,
    B_LOW_QUALITY = 10,
    B_MEDIUM_QUALITY = 50,
    B_HIGH_QUALITY = 100
  };
```

rtm_realloc

Declared in: be/media/RealtimeAlloc.h

```
status_t rtm_realloc(void ** data, size_t new_size)
```

New realtime memory reallocation function.

Constants and Types

Declared in: be/media/MediaDefs.h

- new **B_MEDIA_FLAVORS_CHANGED** value for use as 'what' field of messages.
- new **media_multi_channels** and **media_multi_matrix** enumerations for use with multiple audio channels.
- changes to **media_raw_audio_format** enumeration.
- **media_raw_audio_format** is deprecated. Please use **media_multi_audio_format** instead. **media_format.u.raw_audio** and **media_raw_audio_format::wildcard** have changed to a **media_multi_audio_format** struct, which is upwards compatible.
- new **media_multi_audio_info** struct (used for a new field in **media_encoded_audio_format**)
- new **media_display_flags** struct
- new **media_format_flags** enumeration
- changes to the **media_format** struct, including these new inline functions: **Width()**, **Height()**, and **Encoding()**.

plus changes in constness of existing functions.

- new **B_AVR_FORMAT_FAMILY** in the **media_format_family** typedef

- new structs for use by encoders and file writers: `media_encode_info`, `encode_parameters`, `media_decode_info`

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScreenSaver

Derived from: none
Declared in: be/add-ons/screen_saver/ScreenSaver.h
Library: the **ScreenSaver** preferences application
more...

BScreenSaver provides an interface for developers creating their own screen saver modules. Classes derived from BScreenSaver implement the functions defined here in order to draw on the screen while it's being saved, and to draw into **ScreenSaver**'s *preview screen* (see "Talking to ScreenSaver" for an illustration of the preview screen).

The BScreenSaver class is abstract—it must be subclassed to be used. Furthermore, you never construct BScreenSaver (subclass) objects yourself; they're created automatically as needed by the screen saver engine.

Screen Saver Lifetime

When the screen saver engine wants to use your screen saver to save the screen, it constructs an instance of your BScreenSaver subclass (through the `instantiate_screen_saver()` function, as explained in "Screen Saver Add-ons"), and then calls the object's hook functions, in this order:

- `InitCheck()`; if this returns a value other than `B_OK`, your object is deleted (and the add-on is unloaded).
- `StartSaver()` is called just before your screen saver starts drawing. You implement it to perform any pre-drawing setup.
- `DirectConnected()` is called as soon as your module is in control of the screen, and then periodically whenever the display changes. It's also called when you're about to lose control of the screen.
- `DirectDraw()` draws to the screen by drawing directly to the frame buffer (in the manner of a BDirectWindow). It's called every `TickSize()` microseconds while your saver is in control.
- `Draw()` also draws to the screen, although not directly to the frame buffer. It's also called once every `TickSize()` microseconds. Typically, you implement `DirectDraw()` *or* `Draw()`, but not both. If you do implement both, `DirectDraw()` is called first.
- `StopSaver()` is called when the screen no longer needs to be saved, and just before the screen is restored to its former glory.

When the **ScreenSavers** preferences app loads your module, the series of calls is similar, but with these differences:

- Just after the `StartSaver()` call, you get a `StartConfig()` call that lets you set up **ScreenSavers**' configuration view.
- All drawing is done through `Draw()`—You don't get `DirectConnected()` or `DirectDraw()` calls.
- `StopConfig()`, called just after `StopSaver()`, lets you clean up the configuration view.

The following functions can be called at any time:

- `SupplyInfo()`
- `ModulesChanged()`
- `SaveState()`

What If I Don't Want a Draw() Function?

You don't have to use the `Draw()`/`DirectDraw()` functions; they're provided as a convenience to make simple screen savers easy to write.

One approach to handling your own drawing is to spawn a drawing thread in `StartSaver()` and shut it down in `StopSaver()`. You should implement the `Draw()` function to notify the drawing thread when the screen is available (*frame* is 0). After that, you can handle drawing any way you want.

ScreenSavers' preview monitor won't work properly unless you do your drawing in the `Draw()` function

If you need direct access to the frame buffer, you'll have to handle synchronization with `DirectConnected()` yourself, because that's the only time the screen saver is holding a lock on the display for you (this is handled automatically if you use `DirectDraw()`).

Hook Functions

`InitCheck()`
Can be implemented to let the screen saver engine know if your add-on was initialized properly.

`StartSaver()`
Can be implemented to do things that need to be done just before the screen saver starts drawing.

`StopSaver()`
Can be implemented to do things that need to be done when the screen saver stops drawing.

`Draw()`
Can be implemented to draw the screen saver's entertaining display (to the screen, as well as in the preview displayed by the **ScreenSaver** preferences application).

`DirectConnected()`
Called when the connection to the screen has been made, when the size or format of the frame buffer changes, when the position, size, or shape of the visible part of the content area of the window changes, or when the connection to the screen is terminated.

`DirectDraw()`
Can be implemented to draw the screen saver's entertaining display when you need to directly access the pixels (as in a BDirectWindow).

`StartConfig()`
Can be implemented to display the screen saver's copyright and configuration information in the **ScreenSaver** preferences application.

`StopConfig()`
Can be implemented to handle things that need to be done just before the **ScreenSaver** configuration view goes away.

`SupplyInfo()`
Can be implemented to supply the screen saver system with information about this add-on. The system then passes the info to other screen saver add-ons (by invoking their `ModulesChanged()` functions).

`ModulesChanged()`
Can be implemented to learn about other screen saver add-ons.

`SaveState()`
Can be implemented to save the current state of the screen saver, allowing it to be reconstituted later.

Constructor and Destructor

BScreenSaver()

BScreenSaver(BMessage ***state**, **image_id** *image***)**

Builds a BScreenSaver object from the state information stored in *state* (if any, as explained below). Freshly constructed, your screen saver isn't connected to a BView, so you can't do anything that requires one. There'll be plenty of time for that later, as described in **StartSaver()** and **StartConfig()**. To gauge the success of the construction, the screen saver engine calls **InitCheck()**.

The *state* argument contains whatever items you added when **SaveState()** was invoked on a previous instantiation of your object—in other words, *state* is a record of the state of your object when it last ran. If this is the very first time your screen saver is being constructed, *state* will be empty, and you'll need some sensible default values. The *state* format is up to you—you write it, you read it, nobody else is going to touch it.

The *image* argument is the image (in the Kernel Kit sense) of the add-on itself. This is particularly useful if your screen saver needs to find some resources stored in the add-on. For example, imagine that your BScreenSaver subclass (SpiffySaver) has a BResources *`my_rsrc` member; here's how you'd retrieve the resource:

```
#include <kernel/OS.h>
#include <storage/Resources.h>

SpiffySaver::SpiffySaver( BMessage *info, image_id id )
{
   /* Initialization... */
   ...

   /* Find SpiffySaver's resources so we can use them later. */
   image_info the_info;
   status_t retval = get_image_info( id, &the_info );
   if( retval != B_OK ) {
      /* gripe at the user or bail out. */
   }

   my_rsrc = new BResources;
   retval = my_rsrc->SetTo( new BFile( the_info.name, O_RDONLY ) );
   if( retval != B_OK ) {
      /* gripe at the user or bail out. */
   }

   /*
   * Anything else you need in the constructor, such as
   * finding strings in your resources.
   */
```

```
    . . .
}
```

~BScreenSaver()

```
virtual ~BScreenSaver()
```

Destroys your screen saver; this is what happens when the user wants to get back to digital content creation. By the time the destructor is called, you won't be drawing on the screen anymore, and `StopConfig()` and `StopSaver()` will already have been called.

Member Functions

Draw() , DirectDraw()

```
virtual void Draw(BView *view, int32 frame)
virtual void DirectDraw(int32 frame)
```

These are the hook functions that draw to the screen. For "normal" drawing, you override `Draw()`; if you want to bang the bits directly (such as you would in a BDirectWindow), you should use `DirectDraw()`. In general, you should only implement one function or the other. If you implement both functions, they'll both be called. `DirectDraw()` will always be called before `Draw()` if you implement both of them.

`Draw()` is called once every `TickSize()` microseconds. *view* is the view to draw into; *frame* is a frame counter that starts at 0 and is incremented each time `Draw()` is called. The view starts with a view color of **B_TRANSPARENT_32_BIT**, which means the desktop will be showing through it. If you don't want to see the Desktop (maybe your add-on draws some nice line patterns that look best on a black background), you should clear the view on the first frame (*frame* == 0).

An easy way to clear the view is to set the view color in `StartSaver()` using **BView::SetViewColor()**.

`DirectDraw()`'s protocol (and calling frequency) is the same as for `Draw()`, but, since it draws directly to the screen, `DirectDraw()` lacks a *view* parameter. See `DirectConnected()` for more direct drawing information.

The longer you spend in your `Draw()`/`DirectDraw()` function, the longer the user will have to wait to un-save their system, and that'll annoy everyone. Do your drawing and get out as quickly as possible.

DirectConnected()

```
virtual void DirectConnected(direct_buffer_info *info)
```

Hook function that's called when direct window access is enabled or suspended, or when the state of the display that you'll be (direct) drawing on changes. The *info* argument describes the display state; see the BDirectWindow class (in the Game Kit) for more information about the `direct_buffer_info` structure.

Unlike `BDirectWindow::DirectConnected()`, you don't need to worry about synchronizing with your drawing thread; the screen saver engine handles that for you.

Your DirectConnected() code should be as short as possible, because what it does can affect the performance of the entire system. DirectConnected() should only handle the immediate task of dealing with changes in the direct drawing context, and shouldn't normally do any actual drawing—that's what DirectDraw() is for.

DirectConnected() should only return when it can guarantee that the request specified by *info* will be strictly obeyed.

The structure pointed to by *info* goes away after DirectConnected() returns, so you should cache the information that interests you.

Direct see Draw()

InitCheck()

```
virtual status_t InitCheck(void)
```

Hook function that the screen saver engine uses to assess the success of your object's construction. Your version of InitCheck() should return B_OK if the object constructed itself properly, and B_ERROR if there was a problem. The default InitCheck() returns B_OK.

InitCheck() is called after the constructor, but before the object has been asked to provide any services. Use it to examine the runtime environment, and let the screen saver engine know if you don't like what's going on. For example, if your screen saver requires hardware-accelerated OpenGL, this would be the place to report an error if the hardware can't be found.

LoopOffCount() see SetLoop()
LoopOnCount() see SetLoop()
ModulesChanged() see SupplyInfo()

SaveState()

```
virtual status_t SaveState(BMessage *state) const
```

Implement this hook function to save the current state of your screen saver so it can be reconstituted the next time your object is constructed. You save the state into the *state* message, which will be passed to your object's constructor.

If your version of SaveState() returns anything other than B_OK, the screen saver engine will ignore anything you've put into the *state* message.

SaveState() can be called at any time, and is not guarenteed to be called from StopSaver(); your saver can't depend on it to handle updates to variables being controlled from the

configuration view. You should handle these directly (and immediately) so the user can see their changes happening.

SetLoop() , LoopOnCount() , LoopOffCount()

void SetLoop(int32 *on_count*, **int32** *off_count***)**
int32 LoopOnCount(void) const
int32 LoopOffCount(void) const

These functions are used for controlling animation loop cycles. For some screen savers, it makes sense to draw repeatedly, and then to sleep. For example, a saver to draw ink-blot patterns could draw pixels every frame until enough of them are on the screen, and then spend some time sleeping before clearing the screen and starting over.

Use `SetLoop()` to implement a screen saver that uses a cycle like this. Your `Draw()` and/or `DirectDraw()` method will be called for *on_count* frames (and the frame count will always be 0 to *on_count* - 1), and then the saver engine will sleep (for `TickSize()` microseconds) for *off_count* frames.

For example, if you **`SetLoop( 10, 5 )`**, your saver will draw on frames 0 to 9, and then sleep for the `TickSize()` microseconds 5 times.

The `LoopOnCount()` function returns the current number of drawing frames (the *on_count*).

The `LoopOffCount()` function returns the current number of sleep frames (the *off_count*).

SetTickSize() , TickSize()

void SetTickSize(bigtime_t *size***)**
bigtime_t TickSize(void) const

`SetTickSize()` sets the drawing frequency to *size* microseconds. This lets you control how often your `Draw()` and/or `DirectDraw()` function is called. The default tick size is 50000 (50 milliseconds).

Setting the tick size to 0 causes `Draw()`/`DirectDraw()` to be called with almost no delay; in reality, there will be some variable delay depending on scheduling latency. This is actually a good thing because it gives other threads a chance to run, and also lets the screen saver stay responsive when the user wants to get back to work.

If you set tick size to 0, do *not* call `snooze()`, `snooze_until()`, **`sleep()`** or anything that could block indefinitely. This sort of misbehavior will cause users to delete your screen saver immediately because nobody appreciates a locked up system. If you do need some sort of delay, call `SetTickSize()` before you return from drawing and reset it to 0 during the next frame.

`TickSize()` returns the drawing frequency in microseconds.

StartConfig() , StopConfig()

```
virtual void StartConfig(BView *configView)
virtual void StopConfig(void)
```

Your `StartConfig()` function is called when the **ScreenSaver** preference is about to display your module's configuration view. Any controls that you need to create to let the user fine-tune your module should be added to *configView*. Remember to set the targets of any controls that you add; a good way to do this is to add a custom BView and `SetTarget()` your controls to that custom view in the view's `AttachedToWindow()` function.

Don't draw your screen saver's funky display inside *configView*; it's for displaying information about your module, and controls for fiddling with its settings. `StartSaver()` will be called (with its *preview* argument set to **true**) to give you a chance to show off.

`StopConfig()` is called when the configuration view of the **ScreenSaver** prefs is about to vanish.

StartSaver() , StopSaver()

```
virtual status_t StartSaver(BView *view, bool preview)
virtual void StopSaver(void)
```

An invocation of `StartSaver()` means that the screen saver engine is about to start saving the screen (if *preview* is **false**), or that **ScreenSavers** is about to display the module in its preview monitor (if *preview* is **true**).

In your implementation of StartSaver(), you can't add children to the view, and you can't draw on the screen yet. Wait for calls to `Draw()` or `DirectDraw()` for that. The view will become visible after `StartSaver()` returns.

`StartSaver()` gives you an opportunity to set up your saver because it tells you exactly how large an area you'll be saving via the *view* argument. You can also use *view* to figure out your color space (indirectly, by creating a BScreen with `view->Window()` in the constructor).

Return `B_OK` if all is well, or `B_ERROR` if you're not prepared to draw on the screen. Screen savers with no preview should return `B_ERROR` when asked to provide one:

```
status_t SpiffySaver::StartSaver( BView *view, bool preview )
{
   /* SpiffySaver has no preview... it's not THAT spiffy. */
   if( preview ) return B_ERROR;

   ...
```

```
   return B_OK;
}
```

StopSaver() is called when the screen saver is done displaying. It's called after the very last call to **Draw()** or **DirectDraw()**.

- StopConfig() see StartConfig()
- StopSaver() see StartSaver()

- SupplyInfo() , ModulesChanged()

virtual void SupplyInfo(BMessage *info**) const**
virtual void ModulesChanged(const BMessage *info**)**

These functions aren't currently used by the screen saver system.

These functions are used for communication between screen saver modules.

The screen saver engine calls **SupplyInfo()** when it wants you to supply information about your screen saver. Stuff any information you want other modules to know about into the *info* message. The message is then passed to other screen saver modules through their **ModulesChanged()** functions.

The **ModulesChanged()** hook function is called whenever the list of known screen saver modules changes. The *info* BMessage contains an embedded BMessage for every valid module. Each embedded BMessage is named after its module's add-on, and contains the information that's returned by the module's **SupplyInfo()** implementation.

As an example—although not a very useful one—here we print the name and contents of each embedded message:

```
void SpiffySaver::ModulesChanged( const BMessage *info )
{
   int32 num_addons = info->CountNames( B_MESSAGE_TYPE );

   for( int idx = 0; idx < num_addons; idx ++ ) {
      status_t retval;
      type_code found_type;
      char *name;

      retval = info->GetInfo( B_MESSAGE_TYPE, idx,
                              &name, &found_type );
      if( retval == B_OK ) {
            BMessage msg;
            retval = info->GetMessage( name, &msg );
```

```
            if( retval == B_OK ) {
               printf( "A module named "%s" sent this info:n", name );
            msg.PrintToStream();
            }
      }
   }
}
```

The Be Book Screen Saver Screen Saver Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Screen Saver Table of Contents

- Screen Savers
- Talking to ScreenSaver
- Screen Saver Add-ons
- BScreenSaver
- An Example Screen Saver

The Be Book Table of Contents Screen Saver Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved.

The Be Book Screen Saver

Screen Saver: Master Index

A

An Example Screen Saver	An Example Screen Saver
An Example Screen Saver	An Example Screen Saver

B

Building and Installing your Add-on	Screen Saver Add-ons

C

Constructor and Destructor	BScreenSaver

D

DirectConnected()	BScreenSaver
DirectDraw()	BScreenSaver
Draw()	BScreenSaver

G

Global Functions	Screen Saver Add-ons

H

Hook Functions	BScreenSaver

I

InitCheck()	BScreenSaver
instantiate_screen_saver()	Screen Saver Add-ons

L

LoopOffCount()	BScreenSaver
LoopOnCount()	BScreenSaver

M

Member Functions	BScreenSaver
ModulesChanged()	BScreenSaver

O

openObjectId ScreenSaverAddOn.fm:2 417932	anon
openObjectId ScreenSaverClass.fm:2 418171	anon
openObjectId ScreenSaverExample.fm:2 418052	anon
openObjectId ScreenSaverIntro.fm:2 417932	anon
openObjectId ScreenSaverPref.fm:2 419750	anon

S

SaveState()	BScreenSaver
Screen Saver Add-ons	Screen Saver Add-ons
Screen Saver Add-ons	Screen Saver Add-ons
BScreenSaver	BScreenSaver
BScreenSaver()	BScreenSaver
~BScreenSaver()	BScreenSaver
Screen Saver Lifetime	BScreenSaver
Screen Savers	Screen Savers
Screen Savers	Screen Savers
SetLoop()	BScreenSaver
SetTickSize()	BScreenSaver
StartConfig()	BScreenSaver
StartSaver()	BScreenSaver
StopConfig()	BScreenSaver
StopSaver()	BScreenSaver
SupplyInfo()	BScreenSaver

T

Talking to ScreenSaver	Talking to ScreenSaver
Talking to ScreenSaver	Talking to ScreenSaver
TickSize()	BScreenSaver

W

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

An Example Screen Saver

Here's sample code that implements a screen saver that blanks the screen. This is pretty boring, so you'll probably want to change the **Draw()** function to do something cute.

```
/* Minimal screen saver example. */

#include <add-ons/screen_saver/ScreenSaver.h>
#include <interface/StringView.h>
#include <BeBuild.h>

extern "C" {
_EXPORT
BScreenSaver *instantiate_screen_saver( BMessage *msg, image_id id );
}

class MinimalSaver : public BScreenSaver
{
public:
   MinimalSaver( BMessage *archive, image_id image );
   virtual ~MinimalSaver( void );

   virtual void StartConfig( BView *view );
   virtual status_t StartSaver( BView *view, bool preview );
   virtual void Draw( BView *view, int32 frame );
};

BScreenSaver *instantiate_screen_saver( BMessage *msg, image_id id )
{
   return new MinimalSaver( msg, id );
}

MinimalSaver::MinimalSaver( BMessage *archive, image_id image )
   : BScreenSaver( archive, image )
{
   /* Nothing else to do here... */
}

MinimalSaver::~MinimalSaver( void )
{
   /* Nothing else to do here... */
}

void MinimalSaver::StartConfig( BView *view )
{
   view->AddChild( new BStringView( BRect( 10, 10, 200, 35 ),
```

```
                                        B_EMPTY_STRING,
                                        "Minimal Saver" ) );
}

status_t MinimalSaver::StartSaver( BView *view, bool preview )
{
   /* We're so minimal, we don't have a preview. */
   if( preview ) return B_ERROR;

   return B_OK;
}

void MinimalSaver::Draw( BView *view, int32 frame )
{
   if( frame == 0 ) {
      /* Fill the screen with black on the first frame: */
      view->SetLowColor( 0, 0, 0 );
      view->FillRect( view->Bounds(), B_SOLID_LOW );
   }
}
```


The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Screen Saver Add-ons

Declared in: be/add-ons/screen_saver/ScreenSaver.h
Library: ScreenSaver preferences application
more...

A screen saver add-on embodies a single screen saver module. Each module is represented by a BScreenSaver object. To create this object, the add-on implements and exports the function `instantiate_screen_saver()`. The screen saver system calls the `instantiate_screen_saver()` function when it's time to create and run the add-on's screen saver module.

Building and Installing your Add-on

When you build your add-on, you need to link against the **ScreenSaver** preferences application. To do this you...

- Make a symbolic link to the **ScreenSaver** prefs in your add-on's source directory, and then rename the link to "**_APP_**".
- Add _**APP**_ to the libraries in your BeIDE project file or **Makefile**.

You install your screen saver add-ons in the **Screen Savers** directory in **`B_USER_ADDONS_DIRECTORY`**; you may need to create the **Screen Savers** directory first. (See `find_directory()` in the Storage Kit if you're not familiar with the **`B_USER_ADDONS_DIRECTORY`** constant.)

There's also a **Screen Savers** directory in **`B_BEOS_ADDONS_DIRECTORY`**. This is reserved for screen savers that come with BeOS. Don't put your screen savers in this directory.

Global Functions

instantiate_screen_saver()

BScreenSaver ***instantiate_screen_saver(***BMessage ****msg***, image_id *image*)**

instantiate_screen_saver() must be implemented to create an instance of your BScreenSaver-derived class and return it as a BScreenSaver pointer. *msg* contains the state provided by a previous saver run (via the **SaveState()** function). *image* is the **image_id** of the add-on. You usually just pass these arguments straight into your BScreenSaver subclass' constructor, like this:

```
#include <add-ons/screen_saver/ScreenSaver.h>

class SpiffySaver : public BScreenSaver
{
public:
   SpiffySaver( BMessage *msg, image_id image );
...
};

BScreenSaver *instantiate_screen_saver(BMessage *msg, image_id image)
{
   return new SpiffySaver(msg, id);
}
```

The screen saver system handles bad object allocations for you. This means you don't have to catch **bad_alloc** exceptions in your own code. (If the system is that low on resources, the user has other things to worry about.)

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

- Screen Savers
- Talking to ScreenSaver
- Screen Saver Add-ons
- BScreenSaver
- An Example Screen Saver

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Screen Savers

A Screen Saver is a module that displays entertaining animations whenever your computer has been idle for too long, thus preventing the dreaded "pixel rot" that afflicts so many monitors. The Screen Saver system is made up of these parts:

Part	What is it?
screen_blanker	The application that takes over the screen and runs screen saver modules when it's time to save the screen. It also handles asking for and verifying passwords. **screen_blanker** lives in **/bin**.
screen_saver	An input server add-on that monitors user activity and starts (or stops) **screen_blanker.**
ScreenSaver preferences	The Screen Saver system's preference panel. It loads screen saver modules so the user can configure and preview them.
BScreenSaver class	BScreenSaver provides an interface for developers creating their own screen saver modules. Classes derived from BScreenSaver implement the functions that draw on the screen while it's being saved.
Screen saver add-ons	A screen saver add-on embodies a single screen saver module. It contains the (compiled) BScreenSaver code that you

	write, and provides a hook function, **instantiate_screen_saver()**, that lets the screen saver engine create an instance of your BScreenSaver class. The add-ons are stored in the expected places: the **Screen Savers** directory in **B_BEOS_ADDONS_DIRECTORY** for screen savers that come with BeOS, or **Screen Savers** in **B_USER_ADDONS_DIRECTORY** for ones you've added.

As a screen saver module developer, you have to provide two things: a BScreenSaver subclass, and an implementation of the **instantiate_screen_saver()** function. These responsibilities are discussed in the next two sections ("BScreenSaver" and "Screen Saver Add-ons").

The final two sections take a brief look at the **ScreenSaver** preference application ("Talking to ScreenSaver"), and provide an example of a BScreenSaver class ("An Example Screen Saver").

The Be Book

,

...in lovely HTML...

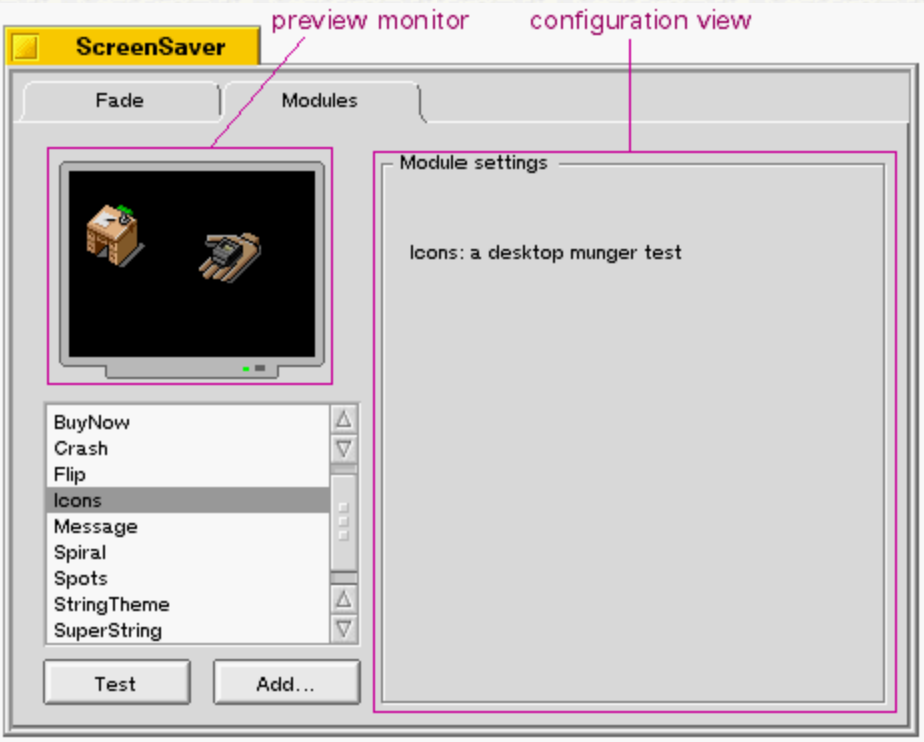
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Talking to ScreenSaver

If you've installed your screen saver module in the right place (see "Building and Installing your Add-on"), the **ScreenSaver** preferences panel will automatically find it and, at the user's request, display a preview of your saver. It does this by loading your add-on and calling your BScreenSaver's hook functions.

As shown below, The panel displays two views that you can draw into: a *preview monitor* that displays a miniature version of your screen saver, and a *configuration view* that you can fill with controls, credits, and so on.



The hook functions that give you access to these views are:

- `StartSaver()` gives you a pointer to the preview monitor. You can initialize the view's state, but you don't draw into it from `StartSaver()`—you have to wait for `Draw()` to be called. (Note that the **ScreenSaver** only calls `Draw()`; you can't use `DirectDraw()` to draw your preview.) The preview is drawn into an offscreen bitmap and then copied to the preview area for added smoothness.

- `StartConfig()` gives you a pointer to the configuration view. You draw into the configuration view directly from within `StartConfig()`.

Note that it's possible for your screen saver module to be loaded by the **ScreenSaver** preferences app *and* ***screen_blanker*** at the same time. Since copies of the add-on are loaded into separate address spaces, you don't normally need to worry about this—unless you're doing something extremely weird.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BScreenSaver Function Summary

Function Summary

Function	Description
DirectConnected()	Hook called when the graphics state changes
DirectDraw()	Hook to update the display directly
Draw()	Hook to update the display
InitCheck()	Returns the status of the most recent initialization
LoopOffCount()	Get animation loop sleep cycles
LoopOnCount()	Get animation loop draw cycles
ModulesChanged()	Hook called to notify this object of other modules
SaveState()	Saves the module's state for later reconstitution
BScreenSaver()	Constructor
~BScreenSaver()	Destructor
SetLoop()	Set animation loop cycles
SetTickSize()	Sets the drawing frequency, in microseconds
StartConfig()	Start drawing the preview display
StartSaver()	Start saving the screen
StopConfig()	Stop drawing the preview display
StopSaver()	Stop saving the screen
SupplyInfo()	Supply the screen saver engine with info about this module

TickSize()	Returns the drawing frequency, in microseconds

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaTrack

Derived from: none
Declared in: be/media/MediaTrack.h **Library:** libmedia.so
Allocation: `BMediaFile::TrackAt()` or `BMediaFile::CreateTrack()`
Summary: *more...*

The BMediaTrack class provides access to a particular track in a media file. It's always instantiated using the `BMediaFile::TrackAt()` or `BMediaFile::CreateTrack()` function.

The BMediaTrack constructor searches for a codec that can handle the encoded data in the track; once that's been done, the track is ready to be used.

If you opened the file for writing, you can write data into the track. If you specified the `B_MEDIA_FILE_REPLACE_MODE` flag when constructing the BMediaFile, you can both read and write from the file. If no decoder is available for the track, you can still use `ReadChunk()` to access the encoded data directly.

After instantiating the BMediaTrack, using the BMediaFile functions for doing so, you should call `InitCheck()` to be sure the track is valid. You can then use `ReadFrames()` and `WriteFrames()` to read and write data to the file, as appropriate. For video data, you should work one frame at a time.

You can also seek particular times or frames using `SeekToTime()` or `SeekToFrame()`.

For an example of how to use BMediaTrack to read and write tracks in media files, see "Reading and Writing Media Files"

As a general rule, you can't use wildcards in any structures used by BMediaTrack functions. You tell BMediaTrack what format you have, and BMediaTrack will simply tell you whether or not that format is supported.

Destructor

~BMediaTrack()

virtual ~**BMediaTrack**(void)

Destructor. You shouldn't delete a BMediaTrack directly; instead, use **BMediaFile::ReleaseTrack()** or **BMediaFile::ReleaseAllTracks()**, or let BMediaFile destroy the track objects when it's deleted.

Member Functions

AddCopyright()

```
status_t AddCopyright(const char *data) const
```

Sets the track's copyright notice to the text specified by *data*, replacing the existing copyright notice if one exists.

The BMediaTrack class doesn't automatically perform any locking to prevent multiple writes to the track from occurring at the same time. If you have multiple threads writing into the same BMediaTrack, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

RETURN CODES

- B_OK. The copyright was set.
- **EPERM**. The file isn't opened for writing.
- Other errors, depending on the codec.

AddTrackInfo()

```
status_t AddTrackInfo(uint32 code, const void *data, size_t size,
    uint32 flags = 0) const
```

Adds an informational record of the specified *type* to the track. The record is specified by the *data* pointer, and is *size* bytes long. *flags* contains flags that modify the operation (none are defined at this time).

The BMediaTrack class doesn't automatically perform any locking to prevent multiple writes to the track from occurring at the same time. If you have multiple threads writing into the same BMediaTrack, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

RETURN CODES

- B_OK. The format was successfully returned.
- Other errors. The error returned depends on the codec interpreting the data.

CountFrames()

int64 **CountFrames**(void) const

Returns the total number of frames in the track.

CurrentFrame()

int64 **CurrentFrame**(void) const

Returns the current frame in the track (the frame that will be read next by **GetNextFrames()**.

CurrentTime()

bigtime_t **CurrentTime**(void) const

Returns the current position within the track, expressed in microseconds since the start of the track.

DecodedFormat()

status_t **DecodedFormat**(media_format **ioFormat*) const

Negotiates the format that the codec will output when decoding the track's data. Pass in *ioFormat* the format that you want (with wildcards as applicable). The codec will find and return in *ioFormat* its best matching format; this format will then be used when outputting decoded data via **ReadFrames()**.

The format is typically of a **B_MEDIA_RAW_AUDIO** or **B_MEDIA_RAW_VIDEO** flavor.

RETURN CODES

- B_OK. The format was successfully negotiated and returned.
- **B_ERROR**. The BMediaTrack wasn't properly initialized, or there's no codec for the track's data.
- Other errors. The error returned depends on the codec interpreting the data.

Duration()

bigtime_t **Duration**(void) const

Returns the total duration of the track, in microseconds.

EncodedFormat()

```
status_t EncodedFormat(media_format *outFormat) const
```

Returns the "native" encoded format of the track's data. This is the format of the data returned by `ReadChunk()`.

RETURN CODES

- B_OK. The format was successfully returned.
- Other errors. The error returned depends on the codec interpreting the data.

FindKeyFrameForTime() , FindKeyFrameForFrame()

```
status_t FindKeyFrameForTime(bigtime_t *inOutTime, int32 flags = 0)
status_t FindKeyFrameForFrame(int64 *inOutFrame, int32 flags = 0)
```

FindKeyFrameForTime() accepts in inOutTime a time, and returns in inOutTime the time at which the closest key frame to that time begins. Likewise, FindKeyFrameForFrame() returns the closest key frame number to the specified frame.

The flags argument lets you indicate whether to seek forward or backward for the key frame.

If you want to find the nearest key frame before the indicated frame or time, specify **B_MEDIA_SEEK_CLOSEST_BACKWARD** in *flags*. If you want to find the nearest key frame after the indicated frame or time, specify **B_MEDIA_SEEK_CLOSEST_FORWARD**.

Flush()

```
status_t Flush(void)
```

Flushes all buffered encoded data to disk; you should call this function after you've finished writing the last frame to the track. This ensures that everything's flushed at the right offset into the file.

RETURN CODES

- B_OK. The buffers were flushed.
- Other errors, depending on the codec.

GetCodecInfo()

```
status_t GetCodecInfo(media_codec_info *codecInfo) const
```

Returns information about the codec being used to read or write the track's data.

RETURN CODES

- B_OK. The codec info was returned.
- **B_BAD_TYPE**. There's no valid codec in use by the track.

GetEncodeParameters() , SetEncodeParameters()

```
status_t GetEncodeParameters(encode_parameters *parameters) const
status_t SetEncodeParameters(encode_parameters *parameters)
```

`GetEncodeParameters()` returns the encode_parameters being used when encoding data on the track.

`SetEncodeParameters()` changes the encode_parameters being used while encoding data.

GetParameterValue() see SetParameterValue()

GetParameterView()

```
BView *GetParameterView(void)
```

Returns a BView containing controls for adjusting the codec's and track's parameters. Returns **NULL** if there isn't a view available.

GetQuality() see SetQuality()

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status code indicating whether or not the track was successfully and completely instantiated.

RETURN CODES

- B_OK. The track was constructed properly.
- **B_BAD_VALUE**. An error occurred setting up the track.

ReadChunk()

```
status_t ReadChunk(char **outBuffer, int32 *ioSize,
```

```
    media_header *header = NULL)
```

Returns in *outBuffer* a pointer to the next *ioSize* bytes of the media track; the actual number of bytes returned is returned in *ioSize*; this may be different if the end of the track is reached. The *header* is set to describe the returned buffer.

The data returned by this function isn't decoded. Typically you'll only use this function if there's no codec available to decode the media data.

RETURN CODES

- B_OK. The format was successfully negotiated and returned.
- **B_ERROR**. The BMediaTrack wasn't properly initialized, or there's no codec for the track's data.
- Other errors. The error returned depends on the codec interpreting the data.

ReadFrames()

```
status_t ReadFrames(void *outBuffer, int64 *outFrameCount,
    media_header *outHeader = NULL)
status_t ReadFrames(void *outBuffer, int64 *outFrameCount,
    media_header *outHeader = NULL,
    media_decode_info *info)
```

Fills the buffer pointed to by *outBuffer* with the next frames or samples from the track, starting at the current position. For video tracks, the next frame of video is decoded and stored into the output buffer. For audio tracks, the buffer is filled with the number of frames negotiated using **DecodedFormat()**. If the end of the track is reached before the buffer is filled, a partial buffer will be returned.

On return, *outFrameCount* indicates the number of frames returned, and *outHeader*, if you specified a non-**NULL** value, contains the header of the buffer containing the frame or frames. You can obtain useful information (such as the media start time for the buffer) from the header.

The second form of this function lets you provide a media_decode_info structure to provide additional information to the decoder, such as how much time it's allowed to use to decode the data and format- and codec-specific information.

RETURN CODES

- B_OK. The frames have been returned.
- **EPERM**. The file wasn't opened for reading.
- Other errors. The error returned depends on the codec interpreting the data.

ReplaceFrames()

```
status_t ReplaceFrames(void *inBuffer, int64 *ioFrameCount,
    media_header *header)
```

Replaces the number of frames specified in *ioFrameCount* in the track. *inBuffer* points to the source buffer for the new frames.

SeekToFrame() , SeekToTime()

```
status_t SeekToFrame(int64 *ioFrame, int32 flags = 0)
status_t SeekToTime(bigtime_t *ioTime, int32 flags = 0)
```

Seeks to the specified position in the track. `SeekToFrame()` accepts a destination position as a frame number, and `SeekToTime()` accepts a destination position as time in microseconds. They each return (in *ioFrame* or in *ioTime*) the position to which they actually moved.

For example, if a video codec is only capable of seeking to key frames, the returned *ioFrame* might be different than the one specified on input.

If you want to seek explicitly to the nearest key frame before the current frame, specify **B_MEDIA_SEEK_CLOSEST_BACKWARD** in *flags*. If you want to find the nearest key frame after the current time, specify **B_MEDIA_SEEK_CLOSEST_FORWARD**.

RETURN CODES

- B_OK. The format was successfully returned.
- Other errors. The error returned depends on the codec interpreting the data.

SetEncodeParameters() see GetEncodeParameters()

SetParameterValue() , GetParameterValue()

```
status_t SetParameterValue(int32 id, const void *value, size_t size)
status_t GetParameterValue(int32 id, const void *value, size_t *size)
```

`SetParameterValue()` sets the value of the parameter specified by *id* to the data pointed to by *value*; this data is *size* bytes long.

`GetParameterValue()` returns in *value* the value of the specified parameter, and the size of the value in bytes in the *size* argument.

RETURN CODES

- B_OK. No error.

- **EPERM**. The codec ins't configurable.
- Other errors, depending on the codec.

SetQuality() , GetQuality()

```
status_t SetQuality(float quality)
status_t GetQuality(float *quality)
```

These functions set and return the codec's quality setting (where 1.0 means maximum quality).

RETURN CODES

- B_OK. No error.
- **B_ERROR**. The codec doesn't support a quality setting.
- Other errors, depending on the codec.

Web()

```
BParameterWeb *Web(void)
```

Returns a BParameterWeb that can be used for configuring the track and codec. Returns **NULL** if the codec doesn't support user configuration.

WriteChunk()

```
status_t WriteChunk(void *data, size_t size, int32 flags = 0)
```

Writes the data pointed to by *data*, which contains *size* bytes of data, into the track. Specify **B_MEDIA_KEY_FRAME** for *flags* if the frame is a key frame. It's assumed that the data is already encoded.

The BMediaTrack class doesn't automatically perform any locking to prevent multiple writes to the track from occurring at the same time. If you have multiple threads writing into the same BMediaTrack, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

In general, you should only use `WriteChunk()` if you're reading compressed data from one file and copying it into another, without trying to interpret the data.

RETURN CODES

- B_OK. The frames have been written.

- **EPERM**. The file wasn't opened for writing.
- Other errors. The error returned depends on the codec writing the data.

WriteFrames()

```
status_t WriteFrames(void *data, int32 numFrames,
    int32 flags = 0)
```

Writes the data pointed to by *data*, which contains *numFrame* frames, into the track. Specify **B_MEDIA_KEY_FRAME** for *flags* if the frame is a key frame.

The BMediaTrack class doesn't automatically perform any locking to prevent multiple writes to the track from occurring at the same time. If you have multiple threads writing into the same BMediaTrack, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

You always have to select an encoder before writing frames into a media track, even if the data is raw audio or video. When writing raw audio or video data into a BMediaTrack, you need to use the raw encoder. Even though this doesn't transform the data, it sets up internal data that's necessary for the file to be played back properly after it's created.

RETURN CODES

- B_OK. The frames have been written.
- **EPERM**. The file wasn't opened for writing.
- Other errors. The error returned depends on the codec writing the data.

Constants

media_seek_type

Declared in: be/media/MediaTrack.h

Constant	Description
`B_MEDIA_SEEK_CLOSEST_FORWARD`	Seek to the nearest key frame after the current position in the track.
`B_MEDIA_SEEK_CLOSEST_BACKWARD`	Seek to the nearest key frame before the current position in the track.
`B_MEDIA_SEEK_DIRECTION_MASK`	Mask the seek flags with this value to obtain the seek direction.

These flags, used when calling `SeekToTime()` and `SeekToFrame()`, let you look for the nearest key frame after or before the current position in the file, without having to guess the time or frame number at which it's located.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFile

Derived from: none
Declared in: be/media/MediaFile.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BMediaFile class represents a file containing media data. When instantiated a BMediaFile with an `entry_ref` to an existing media file, it sniffs the file and figures out the right codecs to use when accessing that file.

To read an existing media file, you can then call `TrackAt()` to instantiate BMediaTrack objects fot the file's tracks; these can in turn be used to decode media data from the file.

You can also write data to the file. In this case, you construct the object by specifying an `entry_ref` and a `media_file_format` specification describing the format of the media data you plan to write into the file. You then call `CreateTrack()` to create each track you want to write into the file. Once each track has been created (but is still empty), you call `CommitHeader()` to write the file's header to disk, and you can use BMediaTrack functions to write the actual media data into the tracks.

Call `CloseFile()` when you're finished writing to it (you don't need to call this if you're reading the file).

For an example of how to use BMediaFile to read and write media files, see "Reading and Writing Media Files"

As a general rule, you can't use wildcards in any structures used by BMediaFile functions. You tell BMediaFile what format you have, and BMediaFile will simply tell you whether or not that format is supported.

Constructor and Destructor

BMediaFile()

```
BMediaFile(const entry_ref *ref)
BMediaFile(BDataIO *source)
BMediaFile(const entry_ref *ref,
    int32 flags = 0)
BMediaFile(BDataIO *source,
    int32 flags = 0)
BMediaFile(const entry_ref *ref,
    const media_file_format *fileFormat,
    int32 flags = 0)
BMediaFile(BDataIO *destination,
    const media_file_format *fileFormat,
    int32 flags = 0)
```

The first four forms of the constructor initializes the BMediaFile to read the file specified by *ref* or by *source*. Once this has been done, be sure to call **InitCheck()** to ensure that the file was successfully sniffed. If sniffing was successful, the BMediaFile object can then be used to instantiate the BMediaTrack objects necessary to read the file's data.

The second two forms prepare a BMediaFile object for writing to a media file, which is specified by either *ref* or *destination*. The format of the file to be written is specified by *fileFormat*.

The flags can be a combination of the following values:

Constant	Description
B_MEDIA_FILE_REPLACE_MODE	Replace any existing media file, if writing.
B_MEDIA_FILE_NO_READ_AHEAD	Don't read ahead in the file.
B_MEDIA_FILE_UNBUFFERED	Use smaller reads. This can be handy if you're trying to stream low-bitrate files withou;waiting for the extractor to fill a large outgoing buffer.
B_MEDIA_FILE_BIG_BUFFERS	Use large buffers.

~BMediaFile()

```
virtual ~BMediaFile(void)
```

Destructor.

Member Functions

AddChunk()

```
status_t AddChunk(int32 type, const void *data, size_t size)
```

Writes a user-defined chunk of data to the file (if doing so is supported by the encoder). The *type* argument indicates the chunk type, *data* points to the data to be written, and *size* indicates the size of the data to write to the file.

The BMediaFile class doesn't automatically perform any locking to prevent multiple writes to the file from occurring at the same time. If you have multiple threads writing into the same BMediaFile, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

RETURN CODES

- B_OK. No error.
- **B_BAD_TYPE**. The BMediaFile doesn't reference a valid file.
- Other errors, depending on the codec.

AddCopyright() , Copyright()

```
status_t *AddCopyright(const char *data)
const char *Copyright(void) const
```

AddCopyright() sets the file's copyright notice to the string specified by *data*. Any existing notice is replaced. The string is copied, so the original belongs to you.

Copyright() returns the media file's copyright notice, or **NULL** if there isn't one. This string belongs to the BMediaFile, so don't delete or change it.

The BMediaFile class doesn't automatically perform any locking to prevent multiple writes to the file from occurring at the same time. If you have multiple threads writing into the same BMediaFile, you must use a locking mechanism (such as a semaphore) to keep writes from overlapping.

RETURN CODES

- B_OK. No error. (**AddCopyright()** only)
- **B_BAD_TYPE**. The BMediaFile doesn't reference a valid file. (**AddCopyright()** only)

- Other errors, depending on the codec. (**AddCopyright()** only)

CloseFile()

```
status_t CloseFile(void)
```

Once all media data has been written to all tracks, you need to call `CloseFile()` so the BMediaFile object can tidy up the file and finish the writing tasks. If you forget to call `CloseFile()`, the resulting file may not be complete.

RETURN CODES

- B_OK. No error.
- **B_BAD_TYPE**. The BMediaFile doesn't reference a valid file.
- Other errors, depending on the codec.

CommitHeader()

```
status_t CommitHeader(void)
```

After you've finished adding new tracks to a file, you need to call `CommitHeader()` to let the BMediaFile set up the file's header. This helps optimize the file writing process.

RETURN CODES

- B_OK. No error.
- **B_BAD_TYPE**. The BMediaFile doesn't reference a valid file.
- Other errors, depending on the codec.

Copyright() see AddCopyright()

CountTracks()

```
int32 CountTracks(void) const
```

Returns the number of tracks in the media file. The return value is undefined if the initialization failed.

CreateTrack()

```
BMediaTrack *CreateTrack(media_format *mediaFormat,
```

const media_codec_info *_codecInfo_**)**
BMediaTrack ***CreateTrack(**media_format *_mediaFormat_**)**

Creates a new track in the file and returns a BMediaTrack object referencing it. The track is configured to use the media format specified by *mediaFormat*, and will be written using the codec described by *codecInfo*. You can only use **BMediaTrack::WriteFrames()** to write into the track.

If you use the second form of the constructor, without a *codecInfo* argument, the track will be written containing raw media data. You can only use **BMediaTrack::WriteChunk()** to write into the track.

The *mediaFormat* indicates the format of the buffers you'll be passing along to the write function, and *codecInfo* specifies the codec you want to use to encode the data into the track.

The *mediaFormat* can't contain any wildcards; you have to specify the exact format that you're going to be providing. *mediaFormat* must also be the same structure you passed to **get_net_encoder()**.

When writing new media files, you should create all your tracks at once before writing any media data. Once all the tracks have been created, be sure to call **CommitHeader()** to write the header to disk. This is necessary since the header size may vary depending on the number of tracks you put in the file, and once you start writing media data into the tracks, it would be difficult (and inefficient) to resize the header.

If an error occurs while creating the new track, **NULL** is returned.

GetFileFormatInfo()

status_t **GetFileFormatInfo(**media_file_format *_info_**)** const

Fills the specified **media_file_format** structure with information describing the file format of the file referenced by the BMediaFile object.

RETURN CODES

- B_OK. No error.
- **B_NO_INIT**. The BMediaFile doesn't reference a valid file.

GetParameterValue() see SetParameterValue()

GetParameterView()

BView ***GetParameterView(**void**)**

Returns a BView containing controls for adjusting the file format's parameters. Returns **NULL** if there isn't a view available.

InitCheck()

```
status_t InitCheck(void) const
```

Returns the result code from the constructor. You should always call this after instantiating a BMediaFile object, but before using it.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_NO_HANDLER**. No codec available for the required or requested format.
- **B_NO_MEMORY**. Not enough memory to set up the codec (writing).
- **B_BAD_INDEX**. The required add-on wasn't found (writing).
- **B_MISSING_SYMBOL**. The add-on doesn't support writing.

ReleaseTrack() , ReleaseAllTracks()

```
status_t ReleaseTrack(BMediaTrack *track)
status_t ReleaseAllTracks(void)
```

`ReleaseTrack()` releases the resources used by the specified *track*. Doing so reduces your application's memory usage. If you want to release every track you're using, you can call `ReleaseAllTracks()`.

Once released, a track can't be used any longer until you use **TrackAt()** again.

If the BMediaFile wasn't properly initialized, or the *index* is invalid, **NULL** is returned.

RETURN CODES

- B_OK. No error.
- **B_ERROR**. Couldn't release the track (it may be invalid, or not actually in use).

SetParameterValue() , GetParameterValue()

```
status_t SetParameterValue(int32 id, const void *value, size_t size)
status_t GetParameterValue(int32 id, const void *value, size_t *size)
```

`SetParameterValue()` sets the value of the parameter specified by *id* to the data pointed to by *value*; this data is *size* bytes long.

`GetParameterValue()` returns in *value* the value of the specified parameter, and the size of the value in bytes in the *size* argument.

RETURN CODES

- B_OK. No error.
- **`B_NO_INIT`**. The BMediaFile doesn't reference a valid file.

TrackAt()

BMediaTrack ***TrackAt**(int32 *index*)*

Returns a BMediaTrack object referencing the track at the specified *index* into the file. The index must be a value between 0 and **`CountTracks()`** - 1.

If the BMediaFile wasn't properly initialized, or the *index* is invalid, **`NULL`** is returned.

You must call **`ReleaseTrack()`** when you're finished with the track.

Web()

BParameterWeb ***Web**(void)*

Returns a BParameterWeb that can be used for configuring the file format's parameters. Returns **`NULL`** if the codec doesn't support user configuration.

Constants

Constructor Flags

Declared in: be/media/MediaFile.h

Constant	Description
`B_MEDIA_FILE_REPLACE_MODE`	Replace any existing media file, if writing.
`B_MEDIA_FILE_NO_READ_AHEAD`	Don't read ahead in the file.
`B_MEDIA_FILE_UNBUFFERED`	Use smaller reads. This can be handy if you're trying to stream low-bitrate files withou;waiting for the extractor to fill a large outgoing buffer.
`B_MEDIA_FILE_BIG_BUFFERS`	Use large buffers.

These flags control the behavior of the constructor.

The Be Book The Media Kit The Media Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFile Function Summary

Function Summary

Function	Description
AddChunk()	Adds user-defined chunks to the file, if this is supported
AddCopyright()	Sets the file's copyright notice text
CloseFile()	Tells the BMediaFile that all track data has been written
CommitHeader()	Tells the BMediaFile that all tracks have been created
Copyright()	Returns the file's copyright notice
CountTracks()	Returns the number of tracks in the media file
CreateTrack()	Creates a new track
GetFileFormatInfo()	Returns information about the underlying file format
GetParameterValue()	Returns the value of the specified parameter
GetParameterView()	Returns a BView containing controls for adjusting the file format's parameters
InitCheck()	Indicates whether or not the object was successfully initialized
BMediaFile()	Constructor
~BMediaFile()	Destructor
ReleaseAllTracks()	Releases all tracks currently in use
ReleaseTrack()	Releases the specified track

SetParameterValue()	Sets the value of the specified parameter
TrackAt()	Returns the specified track
Web()	Returns a BParameterWeb that can be used to configure the file format parameters

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Reading and Writing Media Files

Working with media files becomes relatively painless when you use the BMediaFile and BMediaTrack classes. This section looks at a sample program that converts a media file from one format to another, using these classes.

Preparing a media_format

```
void BuildMediaFormat(int32 width, int32 height, color_space cspace,
            media_format *format) {
   media_raw_video_format *rvf = &format->u.raw_video;

   memset(format, 0, sizeof(*format));

   format->type = B_MEDIA_RAW_VIDEO;
   rvf->last_active = (uint32)(height - 1);
   rvf->orientation = B_VIDEO_TOP_LEFT_RIGHT;
   rvf->pixel_width_aspect = 1;
   rvf->pixel_height_aspect = 3;
   rvf->display.format = cspace;
   rvf->display.line_width = (int32)width;
   rvf->display.line_count = (int32)height;
   if (cspace == B_RGB32)
      rvf->display.bytes_per_row = 4 * width;
   else {
      printf("can't build the format!n");
      exit(5);
   }
}
```

`BuildMediaFormat()` accepts as input parameters describing a video format—the width, height, and color space of the video—and returns a media_format structure describing that format. For our purposes, we require a **`B_RGB32`** colorspace, and the frames will be in raw video format.

Converting the Files

The `transcode()` function below handles actually converting a media file into another format, writing the newly-converted media into a new file. We'll look at it in chunks to lighten the load.

`transcode()` accepts as input a BMediaTrack referring to the video track from the original file, and another BMediaTrack referring to the audio track. *output* is the name of the new file to be created.

family_name specifies the file format family to be used when creating the new file, and *video_name* and *audio_name* indicate by name which encoders should be used for the video and audio tracks.

```
void transcode(BMediaTrack *vidtrack, BMediaTrack *audtrack,
         char *output, char *family_name, char *video_name,
         char *audio_name) {
   char *chunk;
   char *bitmap = NULL, *sound_buffer = NULL;
   bool found_video_encoder = false, found_audio_encoder = false;
   bool found_family;
   int32 i, sz, cookie;
   int64 numFrames, j;
   int64 framesize;
   status_t err;
   entry_ref ref;
   BMediaFile out;
   BMediaTrack vid = NULL, *aud = NULL;
   media_format format, outfmt;
   media_codec_info mci;
   media_file_format mfi;
   media_header mh;

   err = get_ref_for_path(output, &ref);
   if (err) {
      printf("problem with get_ref_for_path() -- %sn", strerror(err));
      return;
   }
```

The function begins by creating an **entry_ref** for the output file. If an error occurs, the function terminates after displaying an error message.

```
   cookie = 0;
   while((err = get_next_file_format(&cookie, &mfi)) == B_OK) {
      if (strcmp(mfi.short_name, family_name) == 0)
         break;
   }
```

```
if (err != B_OK) {
   printf("failed to find a file format handler !n");
   return;
}
```

Next, a loop is used, calling **get_next_file_format()** to find the appropriate handler for the requested file format family. If the specified *family_name* doesn't match the `short_name` field of any of the media file formats available, an error is printed and the function aborts. Otherwise, `mfi` contains a description of the file format.

```
out = new BMediaFile(&ref, &mfi);
err = out->InitCheck();
if (err != B_OK) {
   printf("failed to properly init the output file... (%s)n", strerror(err));
   delete out;
   return;
}
```

Once the appropriate media file format has been determined, a new BMediaFile is created. The `entry_ref` of the output file is specified, and the media file format we selected is provided. This will create the new media file, with no tracks. If this fails (**BMediaFile::InitCheck()** returns something other than **B_OK**), an error message is displayed, the BMediaFile is deleted, and the function returns.

The next chunk of code handles creating the new video track. Note that this only runs if *vidtrack*, the reference to the original BMediaTrack, isn't **NULL**. If it's **NULL**, the file is assumed not to have a video track.

```
if (vidtrack) {
   vidtrack->EncodedFormat(&format);

   if (video_name) {
      int width, height;
      width = format.u.encoded_video.output.display.line_width;
      height = format.u.encoded_video.output.display.line_count;

      memset(&format, 0, sizeof(format));
      BuildMediaFormat(width, height, B_RGB32, &format);

      vidtrack->DecodedFormat(&format);

      bitmap = (char *)malloc(width * height * 4);

      cookie = 0;
      while (get_next_encoder(&cookie, &mfi, &format, &outfmt, &mci) == B_OK) {
         printf("found encoder %s (%d)n", mci.pretty_name, mci.id);
         if (strcmp(video_name, mci.short_name) == 0) {
```

```
            found_video_encoder = true;
            break;
         }
      }
   }
```

If the video track exists, we determine the encoded format by calling **BMediaTrack::EncodedFormat()**. The returned **media_format** describes the format of the video frames in the original file.

If a video encoder for the output was specified, a media_format is constructed using the `BuildMediaFormat()` function we implemented previously. This format will be used for reading the frames from the original file into a raw, unencoded video buffer. We call **BMediaTrack::DecodedFormat()** on the original video track, specifying that this `format` should be used for outputting frames. From this point onward, frames delivered by *vidtrack* will be in raw video format.

A `bitmap` buffer is created. This will contain each frame of raw video while it's being transcoded, after being decoded and before it's encoded into the new file.

Another loop is used to locate an encoder that can convert raw video frames into the desired output format. This is done by calling **get_next_encoder()** in a loop, looking for an encoder that can accept data in the **media_file_format** specified by `mfi`, converting the frames into the media_format specified by `format`. A description of the encoder is returned in `mci`. When a match is found, we check to see if it matches the encoder name, *video_name*, requested by the input arguments. If it does, we accept it by setting the `found_video_encoder` flag to **`true`**.

```
   if (found_video_encoder)
      vid = out->CreateTrack(&format, &mci);
   else
      vid = out->CreateTrack(&format);

   if (vid == NULL) {
      printf("Failed to create video trackn");
      delete out;
      return;
   }
}
```

If a video encoder has been found, a new video track is created, using the found encoder, `mci`. Otherwise, a raw video track is created. This covers the case where the user indicates that he doesn't want any encoding (if they specify a **`NULL`** *video_name*). A pointer to the new BMediaTrack is kept in `vid`.

If the track can't be created, an error message is displayed, the BMediaFile for the output is deleted, and the function returns.

Next, the audio track is prepared in the same way: the encoded format is determined, the decoded format is specified, and an encoder is located, followed by the creation of the new

track. A `sound_buffer` is allocated to contain the amount of audio data that will be stuffed into each buffer returned as the source track is read.

```
if (audtrack) {
   audtrack->EncodedFormat(&format);

   audtrack->DecodedFormat(&format);
   sound_buffer = (char*)malloc(format.u.raw_audio.buffer_size);
   framesize = (format.u.raw_audio.format&15)*format.u.raw_audio.channel_count;

   if (audio_name) {
      cookie = 0;
      while (get_next_encoder(&cookie, &mfi, &format, &outfmt, &mci) == B_OK) {
         printf("found encoder %s (%d)n", mci.pretty_name, mci.id);
         if (strcmp(audio_name, mci.short_name) == 0) {
            found_audio_encoder = true;
            break;
         }
      }
   }

   if (found_audio_encoder)
      aud = out->CreateTrack(&format, &mci);
   else
      aud = out->CreateTrack(&format);

   if (aud == NULL) {
      printf("Failed to create audio trackn");
      delete out;
      return;
   }
}
```

Final touches are then put on the new file's header.

```
// Add the copyright and commit the header
out->AddCopyright("Copyright 1999 Be Incorporated");
out->CommitHeader();
```

In this example, a copyright notice is added to the file by calling **BMediaFile::AddCopyright()**. Once everything's ready, we call **BMediaFile::CommitHeader()** to indicate that we're about to begin writing media data into the file.

Then writing of the video track begins, if there is one.

```
// Process the video track, if any
if (vidtrack) {
```

```
int is_key_frame = 0;

if (found_video_encoder) {
   numFrames = vidtrack->CountFrames();
   for(j = 0; j < numFrames; j++) {
      int64 framecount = 1;
      printf(" r");
      printf("processing frame: %5Ld", j);
      fflush(stdout);
      err = vidtrack->ReadFrames(bitmap, &framecount, &mh);
      if (err) {
         printf("video: GetNextChunk error -- %sn", strerror(err));
         break;
      }
      err = vid->WriteFrames(bitmap, 1, mh.u.encoded_video.field_flags);
      if (err) {
         printf("err %s (0x%x) writing video frame %Ldn",
               strerror(err), err, j);
         break;
      }
   }
```

If there's a video encoder in use (the output data will be encoded), we count up the number of frames in the video track by calling **BMediaTrack::CountFrames()**, then loop over all of the frames.

For each frame, we display a status notice to let the user know what we're doing, then read in the next frame from the source file by calling **BMediaTrack::ReadFrames()**. We specify the `bitmap` buffer as the buffer into which the frame should be read, and `framecount` will contain the number of read frames (it should always be 1 for video tracks). `mh`, a **media_header**, will receive a description of the data in the buffer.

If an error occurs reading the frame, an error message is printed, and the video conversion is terminated.

Otherwise, the output video track, `vid`, is written into by calling **BMediaTrack::WriteFrames()**. The new frame is automatically encoded and appended to the new file. If an error occurs doing this, an error message is displayed and processing is terminated.

If there's no encoder being used (the output file is going to be written as raw or unknown-format data), the following code is used:

```
} else {
   numFrames = vidtrack->CountFrames();
   for(j = 0; j < numFrames; j++) {
      printf(" r");
      printf("processing frame: %5Ld", j);
      fflush(stdout);
```

```
            err = vidtrack->ReadChunk(&chunk, &sz, &mh);
            if (err) {
               printf("video: GetNextChunk error -- %sn", strerror(err));
               break;
            }

            err = vid->WriteChunk(chunk, sz, mh.u.encoded_video.field_flags);
            if (err) {
               printf("err %s (0x%x) writing video frame %Ldn",
                     strerror(err), err, j);
               break;
            }
         }
      }
      printf("r r");
   }
```

The only real difference here is that instead of using **BMediaTrack::ReadFrames()** and **BMediaTrack::WriteFrames()**, we use **BMediaTrack::ReadChunk()** and **BMediaTrack::WriteChunk()**. These work with media data without attempting to interpret the data in any way.

One or the other of these loops will continue until either an error occurs, or the entire video track is converted.

Next, the audio track is converted, if there is one. As you can see, this is done in almost exactly the same manner, except that each buffer we receive will have more than one frame of audio in it. Note how the loop that iterates over the frames adds `framecount` to its counter variable each pass; `framecount` indicates the number of frames returned by the last call to `ReadFrames()` or `ReadChunk()`.

```
   // Process the audio track, if any
   if (audtrack) {
      int64       framecount = 0;

      if (found_audio_encoder) {
         // Decode and encode all the frames
         numFrames = audtrack->CountFrames();
         printf("Total frame count : %Ldn", numFrames);
         for (j = 0; j < numFrames; j+=framecount) {
            err = audtrack->ReadFrames(sound_buffer, &framecount, &mh);
            if (err) {
               printf("video: GetNextChunk error -- %sn", strerror(err));
               break;
            }

            err = aud->WriteFrames(sound_buffer, framecount);
            if (err) {
```

```
            printf("err %s (0x%x) writing audio frame %Ldn",
                  strerror(err), err, j);
            break;
         }
      }
   } else {
      printf("processing chunks...n");
      while (true) {
         err = audtrack->ReadFrames(sound_buffer, &framecount, &mh);
         if (err) {
            printf("audio: GetNextChunk error -- %sn", strerror(err));
            break;
         }

         err = aud->WriteChunk(sound_buffer, framecount*framesize);
         if (err) {
            printf("err %s (0x%x) writing audio chunk %Ldn",
                  strerror(err), err, j);
            break;
         }
      }
   }
   printf("r r");
}
```

Once converting the audio is done, there's very little left to do but release the video and audio buffers we've allocated, close the output BMediaFile, and delete it.

```
   if (bitmap)
      free(bitmap);
   if (sound_buffer)
      free(sound_buffer);

   out->CloseFile();
   delete out;
   out = NULL;
}
```

After this function returns, the caller is responsible for deleting the source BMediaFile.

Using the transcode() Function

Let's look at a `main()` that uses the `transcode()` function to provide a command-line utility for converting media files from one format to another.

```
int main(int argc, char **argv) {
   status_t err;
   entry_ref ref;
   media_format format;
   BMediaFile mediaFile;
   BMediaTrack *track = NULL, *vidtrack = NULL, *audtrack = NULL;
   int32 i, numTracks;
   char *input = NULL, *output = NULL;
   char *video_encoder_name = NULL, *audio_encoder_name = NULL;
   char *family_name = NULL;

   if (argc < 2) {
      printf("usage: %s [-info][-avi|-qt][-wav][-aiff][-v <encoder_name>][-a <encoder_name>] <
[<output>]n", argv[0]);
      return 1;
   }
```

If the number of arguments is less than 2, a usage notice is printed.

```
   for (i=1; i < argc; i++) {
      if (strcmp(&argv[i][0], "-info") == 0) {
         dump_info();
         exit(0);
      } else if (strcmp(&argv[i][0], "-avi") == 0 ||
               strcmp(&argv[i][0], "-wav") == 0 ||
               strcmp(&argv[i][0], "-aiff") == 0 ||
               strcmp(&argv[i][0], "-quicktime") == 0) {
         family_name = &argv[i][1];
      } else if (strcmp(&argv[i][0], "-qt") == 0) {
         family_name = "quicktime";
      } else if (strcmp(&argv[i][0], "-v") == 0 && argv[i+1]) {
         video_encoder_name = argv[i+1];
         i++;
      } else if (strcmp(&argv[i][0], "-a") == 0 && argv[i+1]) {
         audio_encoder_name = argv[i+1];
         i++;
      } else if (input == NULL) {
         input = &argv[i][0];
      } else if (output == NULL) {
         output = &argv[i][0];
      } else {
```

```
        printf("%s: extra argument %sn", &argv[0][0], &argv[i][0]);
    }
}
```

The arguments are interpreted here. The arguments are:

- -info: Dumps information about the available media file formats and encoders; this function can be found in the description of the **`get_next_file_format()`** function.
- -avi: Specifies that the output file should be in AVI format; this sets the `family_name` to "avi".
- -wav: Specifies that the output file should be in WAV format; this sets the `family_name` to "wav".
- -aiff: Specifies that the output file should be in AIFF format; this sets the `family_name` to "aiff".
- -quicktime: Specifies that the output file should be a QuickTime movie; this sets the `family_name` to "quicktime".
- -qt: A shorthand form of -quicktime.
- -v: Lets you specify a video encoder name other than one of the above, such as "-v myformat".
- -a: Lets you specify an audio encoder name other than one of the above, such as "-a audiowonderness".

`input` is the input file's name, and `output` is the name of the new media file to be created.

```
if (output == NULL)
    output = "output";

err = get_ref_for_path(input, &ref);
if (err) {
    printf("problem with get_ref_for_path() -- %sn", strerror(err));
    return 1;
}
```

If no output file name is specified, the name "output" is assumed. An entry_ref to the input file is constructed; if the file isn't found, an error message is printed and the program exits.

```
mediaFile = new BMediaFile(&ref);
err = mediaFile->InitCheck();
if (err) {
    printf("cannot contruct BMediaFile object -- %sn", strerror(err));
    return 1;
}
```

A BMediaFile is then instantiated, referencing the `input` file. If an error occurs, an error is displayed and the program terminates.

```
numTracks = mediaFile->CountTracks();
printf("%s has %d media tracksn", input, numTracks);
const char *copyright = mediaFile->Copyright();
if (copyright)
   printf("#### copyright info: %sn", copyright);
```

The number of tracks in the source file is obtained by calling **BMediaFile::CountTracks()**, and the file's copyright notice is obtained by calling **BMediaFile::Copyright()**. This information is printed for the user to read.

```
for(i=0; i < numTracks; i++) {
   track = mediaFile->TrackAt(i);
   if (!track) {
      printf("cannot get track %d?!?n", i);
      return 1;
   }

   // get the encoded format
   err = track->EncodedFormat(&format);
   if (err) {
      printf("BMediaTrack::EncodedFormat error -- %sn", strerror(err));
      return 1;
   }

   if (format.type == B_MEDIA_RAW_VIDEO ||
      format.type == B_MEDIA_ENCODED_VIDEO) {

      vidtrack = track;
   } else if (format.type == B_MEDIA_RAW_AUDIO ||
            format.type == B_MEDIA_ENCODED_AUDIO) {

      audtrack = track;
   } else {
      mediaFile->ReleaseTrack(track);
      track = NULL;
   }
}
```

Next, a loop iterates over all the tracks in the file by calling **BMediaFile::TrackAt()** to obtain BMediaTrack objects referencing them, one by one. Each track's encoded format is obtained, and is checked to see if the track represents a video or audio track. If it's a video track, it's kept in the `vidtrack` variable. If it's an audio track, it's kept in `audtrack`. If it's neither, the track is released.

This serves to search all the tracks for a video and an audio track to be converted; most movies will only have one of each, but may have other informational tracks, which need to be ignored for our purposes.

```
   if (vidtrack == NULL && audtrack == NULL) {
      printf("%s has no audio or video tracks?!?n", input);
      return 1;
   }
```

If there's neither a video nor an audio track, the source file is empty and isn't worth converting, so an error message is displayed and the program terminates.

```
   if (family_name == NULL && vidtrack == NULL)
      family_name = "wav";
   else if (family_name == NULL)
      family_name = "quicktime";
```

If the user didn't specify an output family, and the file has no video track, the WAV file format is assumed. You can change the program to assume AIFF for audio-only files by changing "wav" to "aiff" here.

For any file that has a video track, if no family is specified by the user, QuickTime format is assumed. Again, if you want the default to be another format, you can change that here.

```
   transcode(vidtrack, audtrack, output, family_name,
            video_encoder_name, audio_encoder_name);
```

The conversion is performed by passing all these parameters to `transcode()` to do the real work.

```
   delete mediaFile;

   return 0;
}
```

Once `transcode()` returns, we simply delete the source file and exit the program. The file has been converted (assuming no errors occurred in `transcode()`.

Integrating Into a Real Application

When creating a real application, you'll need to provide a way for the user to review a list of the various media file formats available, as well as the encoders provided for each format. The `dump_info()` sample function discussed in the description of the **`get_next_file_format()`** function shows how this can be done.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFormats

Derived from: none
Declared in: be/media/MediaFormats.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BMediaFormat class provides a means for translating between BeOS media format identifiers and those used by the codecs themselves. This is primarily useful if you're writing a codec add-on or a file format parser.

Supported Families

The BMediaFormats class currently knows how to translate among the following formats' codec identification methods:

- BeOS
  BeOS uses a constant within the media_format structure to indicate the kind of media being handled.
- QuickTime
  QuickTime uses a two-word vendor/codec pair to indicate the type of media.
- AVI
  The media format is indicated by a one-word constant.
- AVR
  The media format is indicated by a one-word constant.
- ASF
  The media format is indicated by a 128-bit UUID value.
- MPEG
  The media format is specified by a word that indicates the MPEG version and layer.
- AIFF
  The media format is specified by a codec ID word.
- WAV
  The media format is specified by a codec ID word.
- Miscellaneous
  The media format is indicated by a one-word file format value and a one-word codec ID.

Constructor and Destructor

BMediaFormats()

```
BMediaFormats(void)
```

Constructor. Call `InitCheck()` to ensure that the object was initialized properly after you instantiate it.

~BMediaFormats()

```
virtual ~BMediaFormats(void)
```

Destructor.

Member Functions

GetBeOSFormatFor() , GetAVIFormatFor() , GetQuicktimeFormatFor()

```
status_t GetBeOSFormatFor(uint32 format,
    media_format *outFormat,
    media_type type = B_MEDIA_UNKNOWN_TYPE)
```

```
status_t GetAVIFormatFor(uint32 format,
    media_format *outFormat,
    media_type type = B_MEDIA_UNKNOWN_TYPE)
```

```
status_t GetQuicktimeFormatFor(uint32 vendor, uint32 codec,
    media_format *outFormat,
    media_type type = B_MEDIA_UNKNOWN_TYPE)
```

`GetBeOSFormatFor()` returns in *outFormat* a media_format structure that describes the media format corresponding to the given BeOS *format* ID and media_type.

`GetAVIFormatFor()` returns in *outFormat* a media_format structure that describes the media format corresponding to the given AVI *format* ID and media_type.

`GetQuicktimeormatFor()` returns in *outFormat* a media_format structure that describes the media format corresponding to the given QuickTime *vendor* ID, *codec* ID, and media_type.

RETURN CODES

- B_OK. No error.
- **B_BAD_TYPE**. The specified media_type isn't supported.
- **B_MEDIA_BAD_FORMAT**. The specified combination of parameters isn't supported.
- Other errors. The format couldn't be determined.

GetCodeFor()

```
status_t GetCodeFor(const media_format &format,
    media_format_family family,
    media_format_description *outDescription)
```

Given the specified media *format*, returns in *outDescription* the media format description for the specified format *family*.

RETURN CODES

- B_OK. No error.

- **B_MEDIA_BAD_FORMAT**. The specified format isn't recognized.
- Other errors. The code couldn't be determined.

GetFormatFor()

```
status_t GetFormatFor(const media_format_description &description
    media_format *outFormat)
```

Returns in *outFormat* a media_format structure that describes the media format specified by *description*. This lets you easily obtain a media_format structure from file-native description information.

RETURN CODES

- B_OK. No error.
- **B_MEDIA_BAD_FORMAT**. The specified description isn't recognized.
- Other errors. The format couldn't be determined.

GetNextFormat()

```
status_t GetNextFormat(media_format *outFormat,
    media_format_description *outDescription)
```

Fills *outFormat* and *outDescription* with the media_format and media_format_description of the next supported media format.

RETURN CODES

- B_OK. No error.
- **B_BAD_INDEX**. No more formats to return.
- Other errors. The format couldn't be determined.

InitCheck()

```
status_t InitCheck(void)
```

Returns the status code from the constructor; you should call this to be sure the object was properly initialized before issuing any other calls.

RETURN CODES

- B_OK. The object is ready to use.

- **B_BAD_HANDLER**. The object isn't ready.
- Other errors. The Media Server couldn't be contacted.

Lock() , Unlock()

```
bool Lock(void)
void Unlock(void)
```

`Lock()` locks the BMediaFormats object so you can use it with the assurance that it won't change while you're working. You should call this before using any of the other BMediaFormats functions. Returns **true** if it's safe to use the BMediaFormats object; otherwise returns **false**.

`Unlock()` releases the BMediaFormats object when you're done with it; be sure to call it when you've finished your work.

You only need to `Lock()` and `Unlock()` the BMediaFormats object when you're using **RewindFormats()** and **GetNextFormat()**.

MakeFormatFor()

```
status_t MakeFormatFor(const media_format_description *descriptions,
    int32 descCount,
    media_format *ioFormat,
    uint32 flags = 0, void *_reserved = 0)
```

Given the media format descriptions in the *descriptions* array, the number of descriptions in the list, *descCount*, and the media_format *ioFormat*, registers your format description and supported media_format with the Media Kit, and returns a media_format you can use from then on.

This function is called to let the Media Server assign you an encoding value for your media_format_description. If you're implementing a codec add-on or a file parser, you should call this function to let the Media Kit know what you handle.

RETURN CODES

- B_OK. No error.
- **B_BAD_VALUE**. The *descriptions* pointer is **NULL**, the *descCount* is less than 1, or *ioFormat* is **NULL**.
- **B_ERROR**. Some other unfortunate incident occurred.

RewindFormats()

status_t **RewindFormats**(void)

Resets format scanning to the first supported format; this causes **GetNextFormat()** to start at the beginning of the list.

RETURN CODES

- B_OK. The object is ready to use.
- Other errors. Unable to communicate with the Media Server.

Unlock() see Lock()

Constants

make_format_flags

Declared in: be/media/MediaFormats.h

Constant	Meaning
`B_EXCLUSIVE`	Fail if this format has already been registered.
`B_NO_MERGE`	Don't renumber any formats if there are multiple clashing previous registrations, b instead.
`B_SET_DEFAULT`	Set the first format to be the default for the format family when registering more t in the same family. Use only in encoder add-ons.

These constants provide options for how to go about adding new formats.

Media , Format , Families

Declared in: be/media/MediaDefs.h

Constant	Meaning
`B_ANY_FORMAT_FAMILY`	Any family.
`B_BEOS_FORMAT_FAMILY`	BeOS format.
`B_QUICKTIME_FORMAT_FAMILY`	QuickTime format.
`B_AVI_FORMAT_FAMILY`	AVI format.
`B_ASF_FORMAT_FAMILY`	ASF format.
`B_MISC_FORMAT_FAMILY`	Miscellaneous (includes MPEG)

These constants indicate the media format families the BMediaFormats class can map among.

Defined Types

media_aiff_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 codec;
} media_aiff_description;
```

Describes the format of media data as an AIFF codec ID.

media_asf_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   GUID guid;
} media_asf_description;
```

Describes the format of media data as a 128-bit ASF GUID.

media_avi_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 codec;
} media_avi_description;
```

Describes the format of media data as an AVI codec ID.

media_avr_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 id;
} media_avr_description;
```

Describes the format of media data as an AVR codec ID.

media_beos_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   int32 format;
} media_beos_description;
```

Describes the format of media data in a way the BeOS Media Kit understands and appreciates. The *format* field has no predetermined meaning; it's a magic number that only the Media Kit understands.

---

media_codec_info

Declared in: be/media/MediaFormats.h

```
struct encoder_info {
   char pretty_name[96];
   char short_name[32];

   int32 id;
   int32 sub_id;

   int32 pad[63];
};
```

Provides information about an encoder. The `pretty_name` is a complete human-readable name (such as "SuperSqueeze by Applied Interdynamic Systems, Inc."), and the `short_name` is a short-form version of the encoder's name (such as "SuperSqueeze").

The `id` is an opaque ID number that gets passed to **`BMediaFile::AddTrack()`** so the track can identify the encoder.

The padding is reserved space for future expansion of the structure.

---

media_file_format_info

Declared in: be/media/MediaFormats.h

```
struct media_file_format_info {
   char mimetype[64];
   char pretty_name[64];
   char short_name[32];
   char file_extension[8];

   media_format_family family;
```

```
   int64 capabilities;
   int32 id;

   int32 pad[64];
};
```

Describes a media file format. This is used when using BMediaFile objects to determine the type of media file being used.

- `mimetype` is the file format's MIME type string.
- `pretty_name` is a human-readable file format name, such as "QuickTime file".
- `short_name` is a human-readable short name, such as "QuickTime".
- `file_extension` indicates the dot-extension that untyped files of this format might have, such as ".mov".
- `family` indicates the BeOS file format family of the format, such as **`B_MEDIA_QUICKTIME_FAMILY`**.
- `capabilities` contains flags indicating what features the format supports.
- `id` is a unique ID number for the format.
- `pad` pads out the structure, and is reserved.

---

media_format_description

Declared in: be/media/MediaFormats.h

```
typedef struct _media_format_description {
#if defined(__cplusplus)
   _media_format_description();
   ~_media_format_description();
   _media_format_description(const _media_format_description &other);
   _media_format_description & operator=(const _media_format_description
         &other);
#endif
   media_format_family family;
   uint32 _reserved_[3];
   union {
      media_beos_description beos;
      media_quicktime_description quicktime;
      media_avi_description avi;
      media_asf_description asf;
      media_mpeg_description mpeg;
      media_wav_description wav;
```

```
      media_aiff_description aiff;
      media_misc_description misc;
      media_avr_description avr;
      uint32 _reserved_[12];
   } u;
} media_format_description;
```

Describes a media format. The *family* indicates which media family in the union the description uses to identify the format.

---

media_misc_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 file_format;
   uint32 codec;
} media_misc_description;
```

Describes the format of a miscellaneous media file.

---

media_mpeg_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 id;
} media_mpeg_description;
```

Describes the format of media data as an MPEG format ID; this can be any of the following values:

- **B_MPEG_ANY**
  An MPEG format not listed among these choices.
- **B_MPEG_1_AUDIO_LAYER_1**
  MPEG 1 Layer 1 audio format.
- **B_MPEG_1_AUDIO_LAYER_2**
  MPEG 1 Layer 2 audio format.
- **B_MPEG_1_AUDIO_LAYER_3**
  MPEG 1 Layer 3 audio format (commonly known as "MP3").
- **B_MPEG_1_VIDEO**
  An MPEG 1 viedo stream.

---

media_quicktime_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 file_format;
   uint32 codec;
} media_misc_description;
```

Describes the format of media data as a file format/codec pair.

---

media_wav_description

Declared in: be/media/MediaFormats.h

```
typedef struct {
   uint32 codec;
} media_wav_description;
```

Describes the format of media data as a WAV codec ID.

Global C Functions

does_file_accept_format()

```
bool does_file_accept_format(const media_file_format *mfi,
    const media_format *format)
```

Returns **true** if the given media_file_format can contain data in the specified media_format. Otherwise, **false** is returned.

get_next_encoder()

```
status_t get_next_encoder(int32 *cookie,
    media_file_format *mfi,
    media_format *outputFormat,
    media_codec_info *codecInfo)
status_t get_next_encoder(int32 *cookie,
    media_file_format *mfi,
    media_format *inputFormat,
    media_format *outputFormat,
    media_codec_info *codecInfo,
    media_format *acceptedInputFormat,
    media_format *acceptedOutputFormat)
status_t get_next_encoder(int32 *cookie,
    media_codec_info *codecInfo)
```

Given a media_file_format, *mfi*, describing the format of a media file, a media_format, *outputFormat*, describing the desired output format, and optionally a media_format *inputFormat* describing the desired input format, returns in *codecInfo* information about the next encoder capable of handling the format.

You should set the int32 pointed to by *cookie* to 0 initially, and pass the same pointer each time you call `get_next_encoder()`, to scan through all possible encoders. When there are no more encoders left to check, **B_BAD_INDEX** is returned.

If you use the second form of this function, on return the *acceptedInputFormat* and *acceptedOutputFormat* media_format structures will be filled out to indicate the formats that the codec will accept. This form of the function supports wildcards, while the first does not.

The third form of the function is less specific, and can be used to get a complete list of codecs. This can be handy if you're constructing a user interface to allow the user to select the codec they'd like to use, to get a full list of all the possible formats they can create.

RETURN CODES

- B_OK. An encoder was found and returned.
- **B_BAD_INDEX**. The cookie was invalid, or no more encoders found.

See also: **get_next_file_format()** in the global C functions section.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFormats Function Summary

Function Summary

Function	Description
does_file_accept_format()	Reports whether or not a codec supports a particular file format
GetAVIFormatFor()	Returns a media_format structure representing the specified AVI file format
GetBeOSFormatFor()	Returns a media_format structure representing the specified BeOS file format
GetCodeFor()	Returns the format description for the specified format and family
GetFormatFor()	Returns a media_format structure for the specified media_format_description
GetNextFormat()	Returns the next pairing of media_format_description and media_format
GetQuicktimeFormatFor()	Returns a media_format structure representing the specified QuickTime file format
get_next_encoder()	Given a media file format and an output media format, returns information about a codec that can do the job
InitCheck()	Returns the status code from the

	BMediaFormats constructor
Lock()	Locks the BMediaFormats object
MakeFormatFor()	Registers a format, given a media_format_description and media_format, and returns a media_format for it
BMediaFormats()	Constructor
~BMediaFormats()	Destructor
RewindFormats()	Resets format scanning to the first supported format
Unlock()	Unlocks the BMediaFormats object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaTrack Function Summary

Function Summary

Function	Description
AddCopyright()	Sets the track's copyright notice
AddTrackInfo()	Adds information to the track
CountFrames()	Returns the number of frames in the track
CurrentFrame()	Returns the current frame in the track
CurrentTime()	Returns the current position within the track, in microseconds
DecodedFormat()	Negotiates the output format to most closely match your requested format
Duration()	Returns the duration of the track in microseconds
EncodedFormat()	Returns the "native" encoded format of the track's data
FindKeyFrameForFrame()	Returns a key frame as close as possible to the specified time
FindKeyFrameForTime()	Returns a key frame as close as possible to the specified time
Flush()	Flushes buffered encoded data to disk
GetCodecInfo()	Returns information about the codec used for the track's data
GetEncodeParameters()	Returns the current encode

	parameters for the track
GetParameterValue()	Returns the value of the specified parameter
GetParameterView()	Returns a BView containing controls for adjusting the codec's and track's parameters
GetQuality()	Returns the codec's quality setting
InitCheck()	Returns status information indicating whether or not the object was constructed correctly
~BMediaTrack()	Destructor
ReadChunk()	Returns the specified number of bytes of encoded media data
ReadFrames()	Fills a buffer with the next frame (or a group of samples)
ReplaceFrames()	Replaces frames in the track with new data
SeekToFrame()	Seeks to the given frame in the track
SeekToTime()	Seeks to the given time in the track
SetEncodeParameters()	Sets the track's current encode parameters
SetParameterValue()	Sets the value of the specified parameter
SetQuality()	Sets the codec's quality setting
Web()	Returns a BParameterWeb that

	can be used to configure the track's and codec's parameters
WriteChunk()	Writes a chunk of raw data into the file
WriteFrames()	Writes frames into the track

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaEncoder

Derived from: none
Declared in: be/media/MediaEncoder.h **Library:** libmedia.so
Summary: *more...*

The BMediaEncoder class is a base class from which encoder classes may be derived. It handles all the work of locating an appropriate encoder add-on and interfacing with it. All you have to do is implement the `WriteChunk()` and `AddTrackInfo()` functions to write a chunk of encoded data to the destination and to write information about the track into the file.

The BMediaBufferEncoder class is provided for you; it's a derived class that implements `WriteChunk()` to write encoded data into a memory buffer.

An application that wants to encode a stream of data that's not originating from a file should use a subclass of BMediaEncoder (such as BMediaBufferEncoder) to do so.

Hook Functions

AddTrackInfo()

WriteChunk()

Constructor and Destructor

BMediaEncoder()

BMediaEncoder()
BMediaEncoder(const media_format **outputFormat***)** **BMediaEncoder(**const media_codec_info **mci***)**

The constructor sets up the BMediaEncoder. If you use the empty form of the constructor, you'll have to call `SetTo()` to establish the format to be encoded before calling **`encode()`**.

The second form accepts a media_format structure, *inFormat*, that indicates the type of media data that will be input into the encoder.

The third form of the constructor accepts a media_codec_info structure, *mci*, that determines which codec should be used.

If you use either the media_format or media_codec_info form of the constructor, you must call `InitCheck()` to ensure that construction was successful before using any other functions in this class.

~BMediaEncoder()

```
virtual ~BMediaEncoder()
```

Releases the encoder add-on being used by the BMediaEncoder.

Member Functions

AddTrackInfo()

protected: status_t **AddTrackInfo(**uint32 *code*,
const char **data*,
size_t *size***)**

In derived classes, you should implement this function to write track information into the output file. *code* indicates the type of information being added, and *data* is the information data itself. *size* indicates the size of the data to be written.

By default, this function returns `B_ERROR`. Your implementation should return `B_OK` if the information was written successfully; otherwise it should return an appropriate error code.

Encode()

status_t **Encode(**void **inBuffer*,
int64 *frameCount*,
media_encode_info **info***)**

Encodes a chunk of media data from the input buffer by *inBuffer*, which contains *frameCount* frames of data.

The media_encode_info structure *info* is used on input to specify encoding parameters.

After the data is encoded, the encoder will call the derived class' `WriteChunk()` function to write the data into the file.

RETURN CODES

- `B_OK`. No error.
- `B_NO_INIT`. The BMediaEncoder hasn't been initialized.
- Other errors. The encoder's `StartEncoder()` and `Encode()` functions can return errors.

GetEncodeParameters() , SetEncodeParameters()

status_t **GetEncodeParameters(**encode_parameters **inParameters***)** const
status_t **SetEncodeParameters(**encode_parameters **outParameters***)**

`GetEncodeParameters()` returns in inParameters the `encode_parameters` describing how the encoder add-on is configured.

SetEncodeParameters() let you specify the encode_parameters for the add-on.

RETURN CODES

- B_OK. No error.
- Other errors. The encoder's GetEncodeParameters() or SetEncodeParameters() function can return errors.

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t value indicating whether or not construction was successful. You must call this function after construction before calling any other BMediaEncoder functions.

RETURN CODES

- B_OK. The constructor was successful.
- SetTo() errors.

SetFormat()

```
status_t SetInputFormat(media_format *inputFormat,
    media_format *outputFormat,
    media_file_format *mfi = NULL)
```

SetFormat() sets the input and output formats to be used by the encoder, as well as the media_file_format describing the file to which the encoded data will be written.

RETURN CODES

- B_OK. No error.
- B_NO_INIT. The BMediaEncoder hasn't been initialized yet.
- Other errors. The encoder's SetFormat() function can return errors through SetFormat().

SetEncodeParameters() see GetEncodeParameters()

SetTo()

```
status_t SetTo(const media_format *outputFormat)
status_t SetTo(const media_codec_info *mci)
```

SetTo() sets the format of media data that will be encoded by the BMediaEncoder object. This also causes the BMediaEncoder to locate an appropriate codec to use.

The first form accepts a media_format structure, *outputFormat*, that indicates the type of media data that the encoder should output.

The second form of SetTo() accepts a media_codec_info structure, *mci*, that determines which codec should be used.

RETURN CODES

- B_OK. No error.
- B_NO_INIT. The BMediaEncoder isn't initialized.
- get_next_encoder() errors.
- get_image_symbol() errors.

WriteChunk()

```
protected: status_t WriteChunk(const void *chunkData,
    size_t chunkLength,
    media_encode_info *info) = 0
```

In derived classes, you should implement this function to write the chunk of encoded data pointed to by *chunkData* into the destination file or buffer. *chunkLength* indicates the size of the buffer, and *info* provides information about the encoding process that you might need to reference.

Return B_OK if the data is successfully written; otherwise return an appropriate error code.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaEncoder Function Summary

Function Summary

Function	Description
AddTrackInfo()	Hook function to write track information to the file
Encode()	Encodes a chunk of media data
GetEncodeParameters()	Returns an encode_parameters structure specifying how the encoder is configured
InitCheck()	Returns a status_t indicating whether or not the constructor was successful
BMediaEncoder()	Simple constructor
~BMediaEncoder()	Destructor
SetEncodeParameters()	Sets the encode_parameters to be used by the encoder add-on
SetFormat()	Sets the input and output formats
SetTo()	Sets the format to be encoded
WriteChunk()	Hook function to write a chunk of encoded data to the destination

---

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaBufferEncoder

Derived from: BMediaBufferEncoder
Mix-in classes: none
Declared in: be/media/MediaEncoder.h
Library: libmedia.so
Summary: *more...*

The BMediaBufferEncoder class, derived from BMediaBufferEncoder, can be used to encode media data from a memory buffer (instead of from a file).

It implements the **`BMediaBufferEncoder::WriteChunk()`** function to accomplish this task. **`BMediaBufferEncoder::AddTrackInfo()`** is not implemented.

Hook Functions

[WriteChunk()]

Constructor and Destructor

BMediaBufferEncoder()

BMediaBufferEncoder()

```
BMediaBufferEncoder(const media_format *outputFormat)
BMediaBufferEncoder(const media_codec_info *mci)
```

The constructor sets up the BMediaBufferEncoder. If you use the empty form of the constructor, you'll have to call **SetTo()** to establish the format to be encoded before calling **encode()**.

The second form accepts a media_format structure, *inFormat*, that indicates the type of media data that will be input into the encoder.

The third form of the constructor accepts a media_codec_info structure, *mci*, that determines which codec should be used.

If you use either the media_format or media_codec_info form of the constructor, you must call **InitCheck()** to ensure that construction was successful before using any other functions in this class.

~BMediaBufferEncoder()

```
virtual ~BMediaBufferEncoder()
```

Releases the encoder add-on being used by the BMediaBufferEncoder.

Member Functions

EncodeToBuffer()

```
status_t EncodeToBuffer(void *outputBuffer, size_t *outputSize,
    const void *inputBuffer, int64 frameCount,
    media_encode_info *info)
```

Encodes a chunk of media data from the input buffer by *inputBuffer*, which contains *frameCount* frames of data. The encoded data is written into the buffer indicated by *outputBuffer*. On return, *outputSize* indicates how many bytes of data were written into the buffer.

The media_encode_info structure *info* is used on input to specify encoding parameters.

RETURN CODES

- `B_OK`. No error.
- Other errors. The encoder's `Encode()` function can return errors.

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t value indicating whether or not construction was successful. You must call this function after construction before calling any other BMediaBufferEncoder functions.

RETURN CODES

- `B_OK`. The constructor was successful.
- `SetTo()` errors.

WriteChunk()

protected:

```
status_t WriteChunk(const void *chunkData, size_t chunkLength,
      media_encode_info *info) = 0
```

Implemented to write the encoded media data into the buffer specified when `EncodeToBuffer()` was called.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaBufferEncoder Function Summary

Function Summary

Function	Description
EncodeToBuffer()	Encodes a chunk of media data into a memory buffer
InitCheck()	Returns a status_t indicating whether or not the constructor was successful
BMediaBufferEncoder()	Simple constructor
~BMediaBufferEncoder()	Destructor
WriteChunk()	Implementation detail

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaDecoder

Derived from: none
Declared in: be/media/MediaDecoder.h **Library:** libmedia.so
Summary: *more...*

The BMediaDecoder class is a base class from which decoder classes may be derived. It handles all the work of locating an appropriate decoder add-on and interfacing with it. All you have to do is implement the `GetNextChunk()` function to fetch the next chunk of encoded data to be processed.

The BMediaBufferDecoder class is provided for you; it's a derived class that implements `GetNextChunk()` to return data from a memory buffer.

An application that wants to decode a stream of data that's not originating from a file should use a subclass of BMediaDecoder (such as BMediaBufferDecoder) to do so.

Hook Functions

GetNextChunk()

Constructor and Destructor

BMediaDecoder()

BMediaDecoder()

BMediaDecoder(const media_format *inFormat,
const void *info = NULL, size_t infoSize = 0)
BMediaDecoder(const media_codec_info *mci)

The constructor sets up the BMediaDecoder. If you use the empty form of the constructor, you'll have to call SetTo() to establish the format to be decoded before calling Decode().

The second form accepts a media_format structure, *inFormat*, that indicates the type of media data that will be input into the decoder. *info*, if specified, will be filled out with text information about the node; you must specify a buffer *infoSize* bytes long.

The third form of the constructor accepts a media_codec_info structure, *mci*, that determines which codec should be used.

If you use either the media_format or media_codec_info form of the constructor, you must call InitCheck() to ensure that construction was successful before using any other functions in this class.

~BMediaDecoder()

virtual ~BMediaDecoder()

Releases the decoder add-on being used by the BMediaDecoder.

Member Functions

Decode()

status_t **Decode(**void **outBuffer*, int64 **outFrameCount*,
media_header **outMH*, media_decode_info **info***)**

Decodes a chunk of media data into the output buffer specified by *outBuffer*. On return, *outFrameCount* is set to indicate how many frames of data were decoded, and *outMH* is the header for the decoded buffer.

The media_decode_info structure *info* is used on input to specify decoding parameters.

The amount of data decoded is part of the format determined by `SetTo()` or **`SetInputFormat()`.** For audio, it's the buffer_size. For video, it's one frame, which is height*row_bytes. The data to be decoded will be fetched from the source by the decoder add-on calling the derived class' `GetNextChunk()` function.

RETURN CODES

- `B_OK`. No error.
- `B_NO_INIT`. The BMediaDecoder hasn't been initialized.
- Other errors. The decoder's `Decode()` function can return errors.

GetDecoderInfo()

status_t GetDecoderInfo(media_codec_info **outInfo***)** const

Fills out the media_codec_info structure *outInfo* with information about the decoder being used by the BMediaDecoder.

RETURN CODES

- `B_OK`. No error.
- `B_NO_INIT`. The BMediaDecoder hasn't been initialized.
- Other errors. The decoder's `GetCodecInfo()` function can return errors.

GetNextChunk()

protected: status_t **GetNextChunk(**const void ***chunkData*, size_t **chunkLen*,
media_header **mh***)** = 0

In derived classes, you should implement this function to fetch the media data from the source. Set *chunkData* to be a pointer to the next chunk of media data, and *chunkLen* to the size of that buffer. The media_header structure *mh* provides information you can use while fetching the chunk.

This hook is called by the decoder add-on in order to fetch the data from the source.

Return **B_OK** if the chunk is fetched safely, or an appropriate error code otherwise.

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t value indicating whether or not construction was successful. You must call this function after construction before calling any other BMediaDecoder functions.

RETURN CODES

- **B_OK**. The constructor was successful.
- **SetTo()** errors.

SetInputFormat() , SetOutputFormat

```
status_t SetInputFormat(const media_format *inFormat,
    const void *info = NULL, size_t infoSize = 0)
status_t SetOutputFormat(media_format *outputFormat)
```

SetInputFormat() sets the input data format to *inFormat*. On return, *info* (if you don't specify **NULL**) is filled out by the decoder to contain whatever textual information the decoder wants to provide. *infoSize* must indicate the size of the buffer pointed to by *info*.

Unlike **SetTo()**, **SetInputFormat()** function does not select a codec, so the currently-selected codec will continue to be used. You should only use **SetInputFormat()** to refine the format settings if it will not require the use of a different decoder.

SetOutputFormat() sets the format the decoder should output. On return, the *outputFormat* is changed to match the actual format that will be output; this can be different if you specified any wildcards.

RETURN CODES

- **B_OK**. No error.
- **B_NO_INIT**. The BMediaDecoder hasn't been initialized yet.
- Other errors. The decoder's `Sniff()`, `Format()`, and `Reset()` functions can return errors through **SetInputFormat()**.

SetOutputFormat() see SetInputFormat()

SetTo()

```
status_t SetTo(const media_format *inFormat,
    const void *info = NULL, size_t infoSize = 0
status_t SetTo(const media_codec_info *mci)
```

SetTo() sets the format of media data that will be decoded by the BMediaDecoder object. This also causes the BMediaDecoder to locate an appropriate codec to use.

The first form accepts a media_format structure, *inFormat*, that indicates the type of media data that will be input into the decoder. *info*, if specified, will be filled out with text information about the node; you must specify a buffer *infoSize* bytes long.

The second form of SetTo() accepts a media_codec_info structure, *mci*, that determines which codec should be used.

RETURN CODES

- B_OK. No error.
- B_MEDIA_BAD_FORMAT. The specified media_format isn't valid.
- B_BAD_VALUE. Something else went wrong.
- B_ERROR. Unable to instantiate the decoder.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaDecoder Function Summary

Function Summary

Function	Description
Decode()	Decodes a chunk of media data
GetDecoderInfo()	Returns a media_codec_info structure describing the decoder in use
GetNextChunk()	Hook function to fetch the next batch of data to decode
InitCheck()	Returns a status_t indicating whether or not the constructor was successful
BMediaDecoder()	Simple constructor
~BMediaDecoder()	Destructor
SetInputFormat()	Sets the input format
SetTo()	Sets the format to be decoded

The Be Book **The Media Kit** **The Media Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaBufferDecoder

Derived from: BMediaDecoder
Mix-in classes: none
Declared in: be/media/MediaDecoder.h
Library: libmedia.so
Summary: *more...*

The BMediaBufferDecoder class can be used to decode media data from a memory buffer. It implements the `BMediaDecoder::GetNextChunk()` function to fetch data from the memory buffer, and adds a new `DecodeBuffer()` function that decodes data from one buffer into another.

Hook Functions

GetNextChunk()

Constructor and Destructor

BMediaBufferDecoder()

BMediaBufferDecoder()

BMediaBufferDecoder(const media_format *inFormat,
const void *info = NULL, size_t infoSize = 0)
BMediaBufferDecoder(const media_codec_info *mci)

The constructor sets up the BMediaBufferDecoder. If you use the empty form of the constructor, you'll have to call SetTo() to establish the format to be decoded before calling Decode().

The second form accepts a media_format structure, *inFormat*, that indicates the type of media data that will be input into the decoder. *info*, if specified, will be filled out with text information about the node; you must specify a buffer *infoSize* bytes long.

The third form of the constructor accepts a media_codec_info structure, *mci*, that determines which codec should be used.

If you use either the media_format or media_codec_info form of the constructor, you must call InitCheck() to ensure that construction was successful before using any other functions in this class.

~BMediaBufferDecoder()

virtual ~BMediaBufferDecoder()

Releases the decoder add-on being used by the BMediaBufferDecoder.

Member Functions

DecodeBuffer()

```
status_t DecodeBuffer(const void *inputBuffer, size_t inputSize,
    void *outBuffer, int64 *outFrameCount,
    media_header *outMH, media_decode_info *info)
```

Decodes a chunk of media data from the input buffer *inputBuffer* into the output buffer specified by *outBuffer*. The input buffer contains *inputSize* bytes of media data. On return, *outFrameCount* is set to indicate how many frames of data were decoded, and *outMH* is the header for the decoded buffer.

The media_decode_info structure *info* is used on input to specify decoding parameters.

RETURN CODES

- `B_OK`. No error.
- Other errors. The decoder's `Decode()` function can return errors.

GetNextChunk()

protected:

```
status_t GetNextChunk(const void **chunkData, size_t *chunkLen,
    media_header *mh) = 0
```

Implementation detail. This function is implemented to fetch memory from the input buffer specified by the call to `DecodeBuffer()`.

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t value indicating whether or not construction was successful. You must call this function after construction before calling any other BMediaBufferDecoder functions.

RETURN CODES

- `B_OK`. The constructor was successful.
- `SetTo()` errors.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaBufferDecoder Function Summary

Function Summary

Function	Description
DecodeBuffer()	Decodes a chunk of media data from one buffer into another
GetNextChunk()	Hook function to fetch the next batch of data to decode
InitCheck()	Returns a status_t indicating whether or not the constructor was successful
BMediaBufferDecoder()	Simple constructor
~BMediaBufferDecoder()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPushGameSound

Derived from: BStreamingGameSound
Declared in: be/game/PushGameSound.h
Library: libgame.so
Allocation: Constructor only
Summary: *more...*

The BPushGameSound class lets you push buffers of audio data, instead of waiting to be asked for them.

How It Works

The BPushGameSound class uses a single sound buffer, consisting of multiple pages, which play continuously in a loop. Each page is used to construct an audio BBuffer that eventually gets played, and is then recycled and reused again later.

For example, if the sound buffer is 256 kilobytes, and each page is 4 kilobytes, there are 64 pages of audio. When you start the BPushGameSound object, playback begins with the first page. A BBuffer is constructed using that page, then played, and then the buffer is recycled, and the next page is used to create another BBuffer, and so forth. This continues to the 64th page. Once that page is played, playback loops back to the first page again.

Your code pushes audio data into these audio pages. There are two ways you can do this.

Exclusive Access

The first way is to ask the BPushGameSound class to give you a page to fill with audio data. This is done by calling **`LockNextPage()`**. This gives you exclusive access to the next audio page that needs to be filled; you can fill it with whatever sound you want to push, then call **`UnlockPage()`** to release it. It won't be played while it's locked, so you need to stuff your sound into it and release it as quickly as possible.

The Neverending Story

The second way takes better planning, but can give you lower overhead. Call **`LockForCyclic()`** to request access to the entire sound buffer area. This doesn't give you exclusive access, so playback never stops—it keeps looping the entire time, while you write into it. The **`CurrentPosition()`** function tells you where in the buffer area playback is currently occurring.

Your mission (should you choose to accept it) is to stuff audio into the buffer, keeping ahead of this position far enough that playback never catches up to you. As a general rule, you should try to stay at least a page ahead of the current playback position. Keep in mind that when you reach the end of the buffer area, you need to wrap back to the beginning.

This takes more careful effort on your part, but once you have your code properly tuned, you can get very low overhead audio playback this way.

Constructor and Destructor

BPushGameSound()

```
BPushGameSound(size_t inBufferFrameCount,
    const gs_audio_format *format,
    size_t inBufferCount = 2,
    BGameSoundDevice *device = NULL)
```

Prepares the object to play audio pushed by your application. *inBufferFrameCount* specifies the number of frames each audio buffer should be able to hold. *format* indicates the audio format that will be streamed. *inBufferCount* specifies the number of buffers to use, and, as always, *device* is the sound device to use for playback.

Currently, *device* must always be **NULL** to indicate that the default playback device should be used.

By default, two audio buffers are used.

Be sure to call **InitCheck()** before using the BPushGameSound object.

~BPushGameSound

```
virtual ~BPushGameSound()
```

Deletes the semaphore used to lock the object.

Member Functions

CurrentPosition() see LockForCyclic()

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t indicating whether or not the object was successfully initialized. A return value of **B_OK** means everything's fine; any other value means an error occurred in the constructor.

LockForCyclic() , UnlockCyclic() , CurrentPosition()

```
virtual lock_status LockForCyclic(void **outBasePtr, size_t *outSize)
virtual status_t UnlockCyclic(void)
virtual size_t CurrentPosition(void)
```

`LockForCyclic()` gives you access to the entire sound buffer; audio playback continues while you have access. Use `CurrentPosition()` to determine where the playback is currently located in the buffer area, and be sure to stay well ahead of it. See the overview for a more in-depth discussion. If **lock_failed** is returned, you can't access the audio area, but if **lock_ok** is returned, you can start pushing audio.

On return, *outBasePtr* points to the first byte of the audio buffer area, and *outSize* indicates the total size of the audio buffer area.

`UnlockCyclic()` releases the audio area.

RETURN CODES

- B_OK. `UnlockCyclic()` succeeded.
- Other errors may be returned by `UnlockCyclic()` in the future.

UnlockCyclic() see LockForCyclic()

LockNextPage() , UnlockPage()

```
virtual lock_status LockNextPage(void **outPagePtr, size_t *outPageSize)
virtual status_t UnlockPage(void *inPagePtr)
```

`LockNextPage()` requests access to the next page of the sound buffer. The return value is **`lock_ok`** or **`lock_ok_frames_dropped`** if a valid page has been returned. If either of these values is returned, you can then write up to *outPageSize* bytes of audio data into the audio page returned in *outPagePtr*.

If **`lock_failed`** is returned, you can't access an audio page.

`UnlockPage()` releases the audio page pointed to by *inPagePtr*.

RETURN CODES

- B_OK. `UnlockPage()` succeeded.
- Other errors may be returned by `UnlockPage()` in the future.

UnlockPage() see LockNextPage()

Constants

lock_status

Declared in: be/game/PushGameSound.h

Constant	Description
`lock_failed`	Couldn't get the lock; it's not time to update yet.
`lock_ok`	Locked; you can update.
`lock_ok_frames_dropped`	Lccked; you can update, but you may have missed some buffers.

These values are returned by the locking functions in the BPushGameSound class (`LockNextPage()` and `LockForCyclic()`). If either function returns **`lock_failed`**, there isn't a buffer to be filled; your code is pushing buffers too fast.

If **`lock_ok`** is returned, the next buffer is ready. The **`lock_ok_frames_dropped`** result is received if a buffer is available but some buffers have been lost; this can happen if you're not pushing fast enough.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BPushGameSound Function Summary

Function Summary

Function	Description
CurrentPosition()	Returns the current playback position
InitCheck()	Indicates whether or not the object was initialized successfully
LockForCyclic()	Locks the entire sound buffer area for cyclic use
LockNextPage()	Locks the next sound buffer and returns a pointer to it
BPushGameSound()	Constructor
UnlockCyclic()	Releases the cyclic lock
UnlockPage()	Releases the sound buffer previously locked

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Game Kit

The Game Kit provides features that game developers will find particularly useful:

There are two types of functionality provided by the Game Kit:

- Low-level graphics access
- High-performance and convenient sound playback

There's also one global function, `set_mouse_position()`, which lets you move the mouse cursor programatically.

Although designed with games in mind, nothing in the Game Kit is restricted to game applications, except that the user will have to deposit another 50 cents every three minutes.

Low-Level Graphics Access

There are two classes provided for direct access to the underlying graphics hardware: BWindowScreen and BDirectWindow. These two classes both give you direct access to the graphics card's frame buffer; the difference between them is that BWindowScreen always takes over the entire screen, bypassing the Application Server, while BDirectWindow can draw in a window on most graphics hardware.

Although BDirectWindow can do everything BWindowScreen can do, BWindowScreen can be a little easier to use.

High-Performance Audio

Several classes are provided for high-performance audio playback:

- BGameSound
- BFileGameSound
- BSimpleGameSound
- BStreamingGameSound
- BPushGameSound

BGameSound is a base class and isn't used directly; it's the foundation for all the other audio classes in the Game Kit.

BFileGameSound is used to represent a sound effect stored in a disk file, and provides functions for playing the sound. It can be used for playing music loops from audio files, or for playing the occasional large sound effect. BSimpleGameSound, on the other hand, preloads the sound into memory, so that it can be played back with extremely low latency and low overhead (at the expense of memory space).

BPushGameSound are used to let you fill buffers flowing to the speakers (or headphones, or whatever audio output device is being used). Their missions are the same, but their methods are different; BPushGameSound also provides a way to play sound in a cyclic loop, keeping ahead of the

playback point in the buffer. This is the extreme in high-performance, low-latency audio, but does require some extra work on the programmer's part.

BStreamingGameSound requires magic powers to use; its functionality is closely tied to the implementation of the Kit, especially when hardware sound acceleration is involved. So don't touch it, we beg you.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Graphics Card Hook Functions

Declared in: be/add-ons/graphics/GraphicsCard.h
more...

A graphics card driver can implement a set of hook functions that perform specific tasks on behalf of the Application Server and the Game Kit's BWindowScreen object. Drivers should implement as many of these functions as they can.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDirectWindow Function Summary

Function Summary

Function	Description
DirectConnected()	Hook called when the graphics state changes
BDirectWindow()	Constructor
~BDirectWindow()	Destructor
GetClippingRegion()	Returns the current clipping region
IsFullScreen()	Tests whether the window is in full-screen exclusive mode
SetFullScreen()	Enables or disables full-screen exclusive mode
SupportsWindowMode()	Tests if the screen supports window mode

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShape

Derived from: (none)
Mix-in classes: BArchivable
Declared in: be/interface/Shape.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

The BShape class provides a powerful means of representing the outline of any shape that can be comprised of lines or Beziér curves.

The BShapeIterator class provides a means for utilizing BShape objects in your own code. You can also use BShape objects to obtain outlines of characters in a BFont by calling `BFont::GetGlyphShapes()`.

Creating a BShape

A BShape is essentially a list of graphics commands, of which there are four types:

- MoveTo. This sets the BShape's coordinates to a specified point in the shape's space.
- LineTo. This represents a line from the current point to the next point in the shape.
- BeziérTo. This represents a Bezier curve, connecting the current point to a new point, with other points serving to control the shape of the curve.
- Close. This indicates the end of the shape's command list.

Functions by the same names are used to add the corresponding commands to the BShape object. For example, to create a BShape that represents two vertical lines, the following code might be used:

```
BShape shape;
shape.MoveTo(0,0);
shape.LineTo(0,100);
shape.MoveTo(5,0);
shape.LineTo(5,100);
```

Constructor and Destructor

BShape()

BShape( )
BShape(const BShape &*copyFrom*) **BShape(** BMessage ***archive*** **)**

Creates a new BShape object. The first form of the BShape constructor creates an empty shape. The second form, which accepts another BShape as an argument, copies that shape into the new one. The third form reconstructs a BShape from an archive.

~BShape()

virtual ~**BShape( )**

Releases memory occupied by the BShape object.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Returns a new BShape object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* message doesn't contain data for a BShape object, the return value will be `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

AddShape()

```
status_t AddShape(const BShape *otherShape)
```

`AddShape()` adds the lines and curves that comprise the *otherShape* to the shape.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

BezierTo()

```
status_t BezierTo(BPoint controlPoints[3])
```

Adds a command to the BShape that represents a Beziér curve that begins at the current coordinates and is constructed using the specified control points.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

Bounds()

```
BRect Bounds(void) const
```

`Bounds()` returns the bounds rectangle of the entire shape; this rectangle encloses all the lines and Beziér curves that comprise the shape.

Clear()

```
void Clear(void)
```

`Clear()` deletes all the lines and Beziér curves that comprise the shape, leaving it empty.

Close()

void **Close**(void)

Close() should be called when the shape has been fully-constructed by calls to the **BezierTo()**, **LineTo()**, and **MoveTo()** functions.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

LineTo()

status_t **LineTo**(BPoint *point*)

Adds a "draw line" command to the BShape. A line will be drawn from the previous coordinates to the specified coordinates.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

MoveTo()

status_t **MoveTo**(BPoint *point*)

Adds a "move to" command to the BShape. The next **LineTo()** or **BezierTo()** will begin at this point. This lets you create noncontiguous shapes.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

Archived Fields

The `Archive()` function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"pts"	`B_POINT_TYPE`	The list of points used by the shape's commands.
"ops"	`B_INT32_TYPE`	The list of command opcodes indicating the shape's commands.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShape Function Summary

Function Summary

Function	Description
AddShape()	Adds another shape to the shape
BezierTo()	Adds a "beziér" command to the shape
Bounds()	Returns the bounds rectangle of the entire shape
Clear()	Erases the contents of the shape
Close()	Indicates that construction of the shape is complete
Instantiate()	Creates a BShape from a BMessage
LineTo()	Adds a "line" command to the shape
MoveTo()	Adds a "move to coordinates" command to the shape
BShape()	Constructor
~BShape()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShapeIterator

Derived from: (none)
Declared in: be/interface/Shape.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

Applications can create classes derived from BShapeIterator to iterate over the components of a BShape object. There are four virtual functions that the application must implement to process the four graphics commands described in the BShape documentation: `MoveTo()`, `LineTo()`, `BezierTo()`, and `Close()`.

These functions can be implemented to do anything they like with the commands (rendering them to the screen comes to mind as one possibility).

Constructor and Destructor

BShapeIterator()

```
BShapeIterator()
```

Constructor.

~BShapeIterator()

```
virtual ~BShapeIterator()
```

Destructor.

Member Functions

Iterate()

status_t **Iterate(**BShape *_shape_**)**

`Iterate()` iterates over each command that comprises the shape, in order, calling the `Iterate...()` function that corresponds to each command.

RETURN CODES

- B_OK. No error.
- Other errors. None are defined at this time, but you should always check for errors returned by this function.

IterateBezierTo() , IterateClose() , IterateLineTo() , IterateMoveTo()

virtual status_t **IterateBezierTo(**int32 *bezierCount*, BPoint * *bezierPoints***)**
virtual status_t **IterateClose(**void**)**
virtual status_t **IterateLineTo(**int32 *lineCount*, BPoint * *linePoints***)**
virtual status_t **IterateMoveTo(**BPoint * *point***)**

These functions are called by **`Iterate()`** to process the shape commands. Your BShapeIterator-derived class must implement these four functions.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BShapeIterator Function Summary

Function Summary

Function	Description
Iterate()	Iterates over all the components of a BShape
IterateBezierTo()	Hook: process a "beziér" command
IterateClose()	Hook: process a "close" command
IterateLineTo()	Hook: process a "line to" command
IterateMoveTo()	Hook: process a "move to" command
BShapeIterator()	Constructor
~BShapeIterator()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BLocker

Derived from: none
Declared in: be/support/Locker.h
Library: libbe.so
Summary: *more...*

The BLocker class provides a locking mechanism that can be used to protect a section of critical code. The code that you want to protect should be placed between BLocker's **Lock()** and **Unlock()** calls:

```
BLocker *aLock = new BLocker();
. . .
aLock->Lock();
/* Protected code goes here. */
aLock->Unlock();
```

This guarantees that only one thread at a time will pass through the lock. After a thread has locked the BLocker object, subsequent attempts to lock by other threads are blocked until the first thread calls **Unlock()**.

BLocker keeps track of the locking thread—the thread that's currently between **Lock()** and **Unlock()** calls. It lets the thread make nested calls to **Lock()** without blocking. Because of this, you can wrap a BLocker's lock around a series of functions that might, themselves, lock the same BLocker object.

For example, let's say you have a class called BadDog that's declared thus:

```
class BadDog : public BArchivable
{
public:
   void DoThis();
   void DoThat();
   void DoThisAndThat();

private:
   BLocker lock;
};
```

Let's implement the member functions as follows:

```
void BadDog::DoThis()
{
   lock.Lock();
```

```
   /* Do this here. */
   lock.Unlock();
}

void BadDog::DoThat()
{
   lock.Lock();
   /* Do that here. */
   lock.Unlock();
}

void BadDog::DoThisAndThat()
{
   lock.Lock();
   DoThis();
   DoThat();
   lock.Unlock();
}
```

Notice that **`DoThisAndThat()`** wraps the lock around its calls to **`DoThis()`** and **`DoThat()`**, both of which contain locks as well. A thread that gets past the **`Lock()`** call in **`DoThisAndThat()`** won't block when it calls the nested **`Lock()`** calls that it runs into in **`DoThis()`** and **`DoThat()`**.

Keep in mind that nested **`Lock()`** calls must be balanced by an equal number of **`Unlock()`** calls.

See also: the BAutolock class

Constructor and Destructor

BLocker()

```
BLocker(void)
BLocker(const char *name)
BLocker(bool benaphore_style)
BLocker(const char *name, bool benaphore_style)
```

Sets up the object, creating a semaphore to implement the locking mechanism. The optional *name* is for diagnostics and debugging.

By default, a BLocker is implemented as a benaphore –– a hybrid integer mutex and semaphore. This allows for faster performance when there is little contention for the lock. If you'd rather use a pure semaphore for locking, then pass **false** as the *benaphore_style* argument.

~BLocker()

```
virtual ~BLocker()
```

Destroys the lock, deleting the controlling semaphore. If there are any threads blocked waiting to lock the object, they're immediately unblocked.

Member Functions

- CountLocks() see LockingThread()
- CountLockRequests() see LockingThread()
- IsLocked() see LockingThread()

- Lock() , LockWithTimeout() , Unlock()

bool **Lock**(void)
status_t **LockWithTimeout**(bigtime_t *timeout*)
void **Unlock**(void)

Lock() tries to lock the BLocker. The function returns **a)** when the lock is acquired (return = **true**) or **b)** immediately if the BLocker's semaphore is deleted, most commonly as a result of deleting the object (return = **false**). A thread can nest Lock() calls, but each call must be balanced by a concomitant Unlock().

LockWithTimeout() lets you declare a time limit, specified in microseconds. If LockWithTimeout() can't acquire the lock before the time limit expires, it returns B_TIMED_OUT. If the *timeout* is 0, this function immediately returns B_OK (if it locked the BLooper) or B_ERROR (if it failed to obtain the lock). If the *timeout* is B_INFINITE_TIMEOUT, it blocks without limit, just as Lock() does. Note that if Lock() returns 0 (false), it has failed to lock the BLocker, but if LockWithTimeout() returns 0 (B_OK), it has succeeded.

Unlock() releases one level of nested locks and returns immediately. If there are threads blocked waiting for the lock when the lock is released, the thread that's been waiting the longest acquires the lock.

Any thread can call Unlock() on any BLocker—the thread needn't be the lock's current holder. Call IsLocked() before calling Unlock() if you want to make sure you own a lock before you unlock it.

- LockingThread() , IsLocked() , CountLocks() , CountLockRequests() , Sem()

thread_id **LockingThread**(void) const
bool **IsLocked**(void) const
int32 **CountLocks**(void) const
int32 **CountLockRequests**(void) const
sem_id **Sem**(void) const

These functions provide information that may be useful for debugging purposes.

LockingThread() returns the thread that currently has the BLocker locked, or **B_ERROR** if the BLooper isn't locked.

IsLocked() returns `true` if the calling thread currently has the BLocker locked (if it's the locking thread) and `false` if not (if some other thread is the locking thread or the BLocker isn't locked).

CountLocks() returns the number of times the locking thread has locked the BLocker—the number of **Lock()** (or **LockWithTimeout()**) calls that have not yet been balanced by matching **Unlock()** calls.

CountLockRequests() returns the number of threads currently trying to lock the BLocker. The count includes the thread that currently holds the lock plus all threads currently waiting to acquire it.

Sem() returns the **sem_id** for the semaphore that the BLocker uses to implement the locking mechanism.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BLocker Function Summary

Function Summary

Function	Description
CountLockRequests()	Counts the number of pending lock requests
CountLocks()	Counts the number of times the locking thread has acquired the lock
IsLocked()	Tests whether the object is locked
Lock()	Tries to lock the object
LockWithTimeout()	Tries to lock the object within a given time period
BLocker()	Constructor
~BLocker()	Destructor
LockingThread()	Returns the thread_id of the locking thread
Sem()	Returns the locking semaphore
Unlock()	Unlocks the object

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAutolock

Derived from: none
Declared in: be/support/Autolock.h
Library: libbe.so
Summary: *more...*

The BAutolock class provides an easy mechanism for automatically locking and unlocking a BLocker object or, more typically, a BLooper. All you need to do is allocate a BAutolock object on the stack and pass it a target BLocker or BLooper. You may want to make sure the lock is in place before proceeding:

```
BAutolock autolocker(myWindow);
if ( autolocker.IsLocked() ) {
   . . .
}
```

When the stack frame is destroyed, the BAutolock object unlocks the target object and then is itself destroyed.

Constructor and Destructor

BAutolock()

```
inline BAutolock(BLooper *looper)
inline BAutolock(BLocker *locker)
inline BAutolock(BLocker &locker)
```

Locks the target *looper* or *locker* object.

~BAutolock()

```
inline ~BAutolock()
```

Unlocks the target BLooper or BLocker.

Member Functions

IsLocked()

```
inline bool IsLocked(void)
```

Returns `true` if the target BLooper or BLocker is locked, and `false` if not.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAutolock Function Summary

Function Summary

Function	Description
BAutolock()	Constructor
~BAutolock()	Destructor
IsLocked()	Tests whether the target is locked

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BJoystick

Derived from: none
Declared in: be/device/Joystick.h
Library: libdevice.so
Allocation: Constructor only
Summary: *more...*

A BJoystick object provides an interface to a joystick (or other game controller) connected to the computer. The BeOS supports joysticks on the BeBox and Intel platforms.

The BeBox supports up to four analog joysticks, each of which can have up to two axes and two buttons; digital joysticks aren't supported by the built-in game ports. You can install a card that provides additional game ports (such as the Sound Blaster AWE64), and use digital joysticks on those ports.

BeOS for Intel systems supports joysticks through game ports on cards, as well as some built-in game ports.

Unlike the event and message-driven interface to the mouse and keyboard, the interface to a joystick is strictly demand-driven. An application must repeatedly poll the state of the joystick by calling the BJoystick object's `Update()` function. `Update()` queries the port and updates the object's data members to reflect the current state of the joystick.

There are two modes available. Standard mode supports only two axes per joystick, and two buttons per joystick. This mode has been available since the early days of the BeOS. You read the joystick in standard mode by looking at the BJoystick member variables **`vertical`** and **`horizontal`** to determine the joystick's axis values, and **`button1`** and **`button2`** to determine the state of the buttons.

Enhanced mode supports up to 32 buttons per joystick, and an indefinite number of axes and hats (thumb controls, usually located at the top of a stick). It also supports multiple joysticks chained to a single game port. Instead of reading variables to determine the state of the joystick, there are several functions provided to let you do this.

In addition, enhanced mode provides a mechanism for determining what joysticks are available and what types of (and how many) controls are available on the joysticks. There's also a preference application (cleverly named "Joysticks") that lets the user select and configure joysticks connected to their computer.

Data Members

bigtime_t **`timestamp`**
: The time of the most recent update, as measured in microseconds from the beginning of 1970.

int16 **`horizontal`**
: The horizontal position of the joystick at the time of the last update. Values increase as the joystick moves from left to right.

int16 **`vertical`**
: The vertical position of the joystick at the time of the last update. Values decrease as the joystick moves forward and increase as it's pulled back.

bool **`button1`**
: `false` if the first button was pressed at the time of the last update, and `true` if not.

bool **`button2`**
: `false` if the second button was pressed at the time of the last update, and `true` if not.

The **`horizontal`** and **`vertical`** data members record values read directly from the ports, values that simply digitize the analog output of the joysticks. This class makes no effort to translate the values to a standard scale or range. Values can range from 0 through 4,095, but joysticks typically don't use the full range and some don't register all values within the range that is used. The scale is not linear—identical increments in different parts of the range can reflect differing amounts of horizontal and vertical movement. The exact variance from linearity and the extent of the usable range are partly characteristics of the individual joystick and partly functions of the computer's hardware.

Typically you won't use these data members if you're using enhanced mode; they're provided primarily for backward compatibility.

Constructor and Destructor

BJoystick()

```
BJoystick(void)
```

Initializes the BJoystick object so that all values are set to 0. Before using the object, you must call `Open()` to open a particular joystick port. For the object to register any meaningful values, you must call `Update()` to query the open port.

See also: `Open()`, `Update()`

~BJoystick()

```
virtual ~BJoystick(void)
```

Closes the port, if it was not closed already.

Member Functions

- ButtonValues() see CountButtons()
- Close() see Open()

- CountAxes() , GetAxisValues()

int32 **CountAxes**(void)
status_t **GetAxisValues**(int16 **outValues*, int32 *forStick* = 0)

`CountAxes()` returns the number of axes on the opened game port.

`GetAxisValues()` fills the array pointed to by *outValues* with the values of all the axes connected to the specified joystick. The *forStick* parameter lets you choose which joystick on the game port to read the values from (the default is to read from the first stick on the port).

The returned values range from -32,768 to 32,767.

The array *outValues* must be large enough to contain the values for all the axes. You can ensure that this is the case by using the following code:

```
int16 *axes;

axes = (int16 *) malloc(sizeof(int16) * stick->CountAxes());
stick->GetAxisValues(axes);
```

These functions can only be used in enhanced mode.

RETURN CODES

- `B_OK`. The name was returned successfully.
- `B_BAD_VALUE`. The joystick specified by *forStick* doesn't exist.

- CountButtons() , GetButtonValues()

int32 **CountButtons**(void)
uint32 **ButtonValues**(void)

`CountButtons()` returns the number of buttons on the opened game port.

`ButtonValues()` returns a 32-bit number in which each bit represents the state of one button. The *forStick* parameter lets you choose which joystick on the game port to read the values

from (the default is to read from the first stick on the port). You can deterimine if a particular button is down by using the following code:

```
uint32 buttonValues = stick->ButtonValues();

if (buttonValues & (1 << whichButton)) {
   /* button number whichButton is pressed */
}
```

These functions can only be used in enhanced mode.

CountDevices() , GetDeviceName()

```
int32 CountDevices(void)
status_t GetDeviceName(int32 index, char *outName,
    size_t bufSize = B_OS_NAME_LENGTH)
```

`CountDevices()` returns the number of game ports on the computer.

`GetDeviceName()` returns the name of the device specified by the given *index*. The buffer pointed to by *outName* is filled with the device name; *bufSize* indicates the size of the buffer.

The names returned by `GetDeviceName()` can be passed into the `Open()` function to open a device.

The BJoystick doesn't need to have an open device before you use these functions; in fact, your application will typically use these to provide user interface allowing the user to choose the joystick device they'd like to use.

RETURN CODES

- `B_OK`. The name was returned successfully.
- `B_BAD_INDEX`. The specified *index* is invalid.
- `B_NAME_TOO_LONG`. The device name is too long for the buffer.

CountHats() , GetHatValues()

```
int32 CountHats(void)
status_t GetHatValues(int8 *outHats, int32 forStick = 0)
```

`CountHats()` returns the number of hats on the opened game port.

`GetHatValues()` fills the array pointed to by *outHats* with the values of all the hats connected to the specified joystick. The *forStick* parameter lets you choose which joystick on the game port to read the values from (the default is to read from the first stick on the port).

The return value means the following:

Value	Meaning
0	Centered
1	Up
2	Up and right
3	Right
4	Down and right
5	Down
6	Down and left
7	Left
8	Up and left

The array *outHats* must be large enough to contain the values for all the hats. You can ensure that this is the case by using the following code:

```
int8 *hats;

hats = (int8 *) malloc(sizeof(int8) * stick->CountAxes());
stick->GetHatValues(hats);
```

These functions can only be used in enhanced mode.

RETURN CODES

- B_OK. The name was returned successfully.
- **B_BAD_VALUE**. The joystick specified by *forStick* doesn't exist.

CountSticks()

```
int32 CountSticks(void)
```

Returns the number of joysticks connected to the opened game port.

This function can only be used in enhanced mode.

GetAxisNameAt() , GetHatNameAt() , GetButtonNameAt()

status_t **GetAxisNameAt(**int32 *index*, BString ***outName***)
status_t **GetHatNameAt(**int32 *index*, BString ***outName***)
status_t **GetButtonNameAt(**int32 *index*, BString ***outName***)

Returns the name of the control specified by the given *index*. The BString object pointed to by *outName* is set to the control's name.

`GetAxisNameAt()` returns the specified axis' name, `GetHatNameAt()` returns the specified hat's name, and `GetButtonNameAt()` returns the specified button's name.

These functions can only be used in enhanced mode.

RETURN CODES

- **B_OK**. The name was returned successfully.
- **B_BAD_INDEX**. The specified *index* is invalid.

- GetAxisValues() see CountAxes()
- GetButtonNameAt() see GetAxisNameAt()

- GetControllerModule() , GetControllerName()

status_t **GetControllerModule(**BString ***outName***)
status_t **GetControllerName(**BString ***outName***)

`GetControllerModule()` returns the name of the joystick module that represents the opened joystick device. If the device isn't in enhanced mode, this always returns "Legacy".

`GetControllerName()` returns the name of the joystick that's been configured for the opened device. This is the same string that appears in the Joysticks preference application. The returned string is always "2-axis" if the device isn't in enhanced mode.

These functions can only be used in enhanced mode.

RETURN CODES

- B_OK. The name was returned successfully.
- **B_BAD_INDEX**. The specified *index* is invalid.

- GetDeviceName() see CountDevices()
- GetHatNameAt() see GetAxisNameAt()
- GetHatValues() see CountHats()

EnableCalibration() see IsCalibrationEnabled()

EnterEnhancedMode()

bool **EnterEnhancedMode(**const entry_ref **ref* = NULL**)**

Switches the device into enhanced mode. If *ref* isn't **NULL**, it's treated as a reference to a joystick description file (such as those in /boot/beos/etc/joysticks). If *ref* is **NULL**, the currently-configured joystick settings (as per the Joysticks preference application) are used.

If enhanced mode is entered successfully (or the device is already in enhanced mode), **true** is returned. Otherwise, `EnterEnhancedMode()` returns **false**.

IsCalibrationEnabled() , EnableCalibration()

bool **IsCalibrationEnabled(**void**)**
status_t **EnableCalibration(**bool *calibrates* = true**)**

`IsCalibrationEnabled()` returns **true** if axis values returned by the joystick will be calibrated automatically into the range -32,768 to 32,767, or **false** if the raw values will be returned.

`EnableCalibration()` enables or disables automatic calibration for the joystick's axes. Specify a value of **true** for *calibrates* to enable calibration; otherwise, specify **false**.

The names returned by `GetDeviceName()` can be passed into the **Open()** function to open a device.

The Joysticks preference application lets the user calibrate the joystick. Calibration is enabled by default. These functions may only be used in enhanced mode.

RETURN CODES

- B_OK. The name was returned successfully.
- **B_NO_INIT**. The device isn't in enhanced mode.

Open() , Close()

status_t **Open(**const char **devName***)** status_t **Open(**const char **devName*, bool *enterEnhanced***)**
void **Close(**void**)**

These functions open the joystick port specified by *devName* and close it again. The *devName* should be the name of a game port device. The easiest way to determine valid device names

is by using the CountDevices() and GetDeviceName() functions.

If it's able to open the port, Open() returns a positive integer. If unable or if the *name* isn't valid, it returns B_ERROR. If the *name* port is already open, Open() tries to close it first, then open it again.

By default, Open() opens the device in enhanced mode. If you want to use the old, unenhanced mode, you can use the second form of Open() to specify whether or not you want to use enhanced mode; set *enterEnhanced* to **true** to use enhanced mode; specify **false** if you don't want enhanced mode.

Even in enhanced mode, the classic BJoystick data members are valid (for compatibility with older applications). However, they only give you access to the first two axes and buttons of the joystick.

To be able to obtain joystick data, a BJoystick object must have a port open.

SetMaxLatency()

status_t **SetMaxLatency(**bigtime_t *maxLatency***)**

Specifies the maximum latency to allow when accessing the joystick. Returns B_OK if the change is applied successfully, otherwise returns an error code.

Update()

status_t **Update(**void**)**

Updates the data members of the object so that they reflect the current state of the joystick. An application would typically call Update() periodically to poll the condition of the device, then read the values of the data members, or—if in enhanced mode—call the various functions for obtaining joystick state information.

This function returns B_ERROR if the BJoystick object doesn't have a port open, and B_OK if it does.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Device Kit Table of Contents

- The Device Kit
- BJoystick
- BSerialPort

The Be Book **The Device Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Device Kit: Master Index

C

ClearInput()	BSerialPort
ClearOutput()	BSerialPort
Close()	BJoystick
Close()	BSerialPort
Constructor and Destructor	BJoystick
Constructor and Destructor	BSerialPort
CountAxes()	BJoystick
CountButtons()	BJoystick
CountDevices()	BJoystick
CountHats()	BJoystick
CountSticks()	BJoystick

D

DataBits()	BSerialPort
Data Members	BJoystick
DataRate()	BSerialPort
data_bits	BSerialPort
data_rate	BSerialPort
The Device Kit	The Device Kit
The Device Kit	The Device Kit

E

EnableCalibration()	BJoystick
EnterEnhancedMode()	BJoystick

F

FlowControl()	BSerialPort

G

GetAxisNameAt()	BJoystick
GetAxisValues()	BJoystick
GetButtonNameAt()	BJoystick
GetButtonValues()	BJoystick
GetControllerModule()	BJoystick
GetControllerName()	BJoystick
GetDeviceName()	BJoystick
GetHatNameAt()	BJoystick
GetHatValues()	BJoystick

H

B_HARDWARE_CONT	BSerialPort

I

IsCTS()	BSerialPort
IsCalibrationEnabled()	BJoystick
IsDCD()	BSerialPort
IsDSR()	BSerialPort
IsRI()	BSerialPort

J

BJoystick	BJoystick
BJoystick()	BJoystick
~BJoystick()	BJoystick

M

Member Functions	BJoystick
Member Functions	BSerialPort

O

Open()	BJoystick
Open()	BSerialPort

P

ParityMode()	BSerialPort
parity_mode	BSerialPort

R

Read()	BSerialPort

S

BSerialPort	BSerialPort
BSerialPort()	BSerialPort
~BSerialPort()	BSerialPort
SetBlocking()	BSerialPort
SetDTR()	BSerialPort
SetDataBits()	BSerialPort
SetDataRate()	BSerialPort
SetFlowControl()	BSerialPort
SetMaxLatency()	BJoystick
SetParityMode()	BSerialPort
SetRTS()	BSerialPort
SetStopBits()	BSerialPort
SetTimeout()	BSerialPort
B_SOFTWARE_CONT	BSerialPort
StopBits()	BSerialPort
stop_bits	BSerialPort

U

Update()	BJoystick

W

WaitForInput()	BSerialPort
Write()	BSerialPort

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSerialPort

Derived from: *none*
Declared in: be/device/SerialPort.h
Library: libdevice.so
more...

A BSerialPort object represents an RS-232 serial connection to the computer. Through BSerialPort functions, you can read data received at a serial ports and write data over the connection. You can also configure the connection—for example, set the number of data and stop bits, determine the rate at which data is sent and received, and select the type of flow control (hardware or software) that should be used.

To read and write data, a BSerialPort object must first open one of the serial ports by name. To find the names of all the serial ports on the computer, use the `CountDevices()` and `GetDeviceName()` functions:

```
BSerialPort serial;
char devName[B_OS_NAME_LENGTH];
int32 n = 0;

for (int32 n = serial.CountDevices()-1; n >= 0; n--) {
   serial.GetDeviceName(n, devName);

   if ( serial.Open(devName) > 0 )
      ....
}
```

The BSerialPort object communicates with the driver for the port it has open. The driver maintains an input buffer to collect incoming data and a smaller output buffer to hold outgoing data. When the object reads and writes data, it reads from and writes to these buffers.

The serial port drivers, and therefore BSerialPort objects, send and receive data asynchronously only.

Constructor and Destructor

BSerialPort()

```
BSerialPort(void)
```

Initializes the BSerialPort object to the following default values:

- Hardware flow control (see `SetFlowControl()`)
- A data rate of 19,200 bits per second (see `SetDataRate()`)
- A serial unit with 8 bits of data, 1 stop bit, and no parity (see `SetDataBits()`)
- Blocking with no time limit—an infinite timeout—for reading data (see `Read()`)

The new object doesn't represent any particular serial port. After construction, it's necessary to open one of the ports by name.

The type of flow control must be decided before a port is opened. But the other default settings listed above can be changed before or after opening a port.

See also: `Open()`

~BSerialPort()

```
virtual ~BSerialPort()
```

Makes sure the port is closed before the object is destroyed.

Member Functions

ClearInput() , ClearOutput()

```
void ClearInput(void)
void ClearOutput(void)
```

These functions empty the serial port driver's input and output buffers, so that the contents of the input buffer won't be read (by the `Read()` function) and the contents of the output buffer (after having been written by `Write()`) won't be transmitted over the connection.

The buffers are cleared automatically when a port is opened.

See also: `Read()`, `Write()`, `Open()`

Close() see Open()

DataBits() see SetDataBits()

DataRate() see SetDataRate()

FlowControl() see SetFlowControl()

IsCTS()

```
bool IsCTS(void)
```

Returns `true` if the Clear to Send (CTS) pin is asserted, and `false` if not.

IsDCD()

```
bool IsDCD(void)
```

Returns `true` if the Data Carrier Detect (DCD) pin is asserted, and `false` if not.

IsDSR()

```
bool IsDSR(void)
```

Returns `true` if the Data Set Ready (DSR) pin is asserted, and `false` if not.

IsRI()

bool **IsRI**(void)

Returns `true` if the Ring Indicator (RI) pin is asserted, and `false` if not.

Open() , Close()

status_t **Open**(const char **name*)
void **Close**(void)

These functions open the *name* serial port and close it again. To get a serial port name, use the GetDeviceName() function; an example is shown in the introduction.

To be able to read and write data, the BSerialPort object must have a port open. It can open first one port and then another, but it can have no more than one open at a time. If it already has a port open when **Open()** is called, that port is closed before an attempt is made to open the *name* port. (Thus, both **Open()** and **Close()** close the currently open port.)

Open() can't open the *name* port if some other entity already has it open. (If the BSerialPort itself has *name* open, **Open()** first closes it, then opens it again.)

When a serial port is opened, its input and output buffers are emptied and the Data Terminal Ready (DTR) pin is asserted.

RETURN CODES

positive integers (not 0). Success.

- **B_PERMISSION_DENIED. The port is already open.**
- **B_ERROR**. The port couldn't be opened for some other reason.

ParityMode() see SetDataBits()

Read() , SetBlocking() , SetTimeout()

ssize_t **Read**(void **buffer*, size_t *maxBytes*)
void **SetBlocking**(bool *shouldBlock*)
status_t **SetTimeout**(bigtime_t *timeout*)

Read() takes incoming data from the serial port driver and places it in the data *buffer* provided. In no case will it read more than *maxBytes*—a value that should reflect the capacity of the *buffer*. The input buffer of the driver, from which **Read()** takes the data, holds a maximum of 2,024 bytes (2048 on Mac hardware). This function fails if the BSerialPort object doesn't have a port open.

The number of bytes that Read() will read before returning depends not only on *maxBytes*, but also on the *shouldBlock* flag and the *timeout* set by the other two functions.

- SetBlocking() determines whether Read() should block and wait for *maxBytes* of data to arrive at the serial port if that number isn't already available to be read. If the *shouldBlock* flag is `true`, Read() will block. However, if *shouldBlock* is `false`, Read() will take however many bytes are waiting to be read, up to the maximum asked for, then return immediately. If no data is waiting at the serial port, it returns without reading anything.
- The default *shouldBlock* setting is `true`.
- SetTimeout() sets a time limit on how long Read() will block while waiting for data to arrive at the port's input buffer. The *timeout* is relevant to Read() only if the *shouldBlock* flag is `true`. However, the time limit also applies to the WaitForInput() function, which always blocks, regardless of the *shouldBlock* setting.
- There is no time limit if the *timeout* is set to B_INFINITE_TIMEOUT—Read() (and WaitForInput()) will block forever. Otherwise, the *timeout* is expressed in microseconds and can range from a minimum of 100,000 (0.1 second) through a maximum of 25,500,000 (25.5 seconds); differences less than 100,000 microseconds are not recognized; they're rounded to the nearest tenth of a second.
- The default *timeout* is B_INFINITE_TIMEOUT.

RETURN CODES

Read() returns...

- *non-negative integer*. Success; the value is the number of bytes that were read.
- B_INTERRUPTED. The operation was interrupted by a signal.
- B_FILE_ERROR. The BSerialPort doesn't have a port open.
- SetTimeout() returns...
- B_OK. Success.
- B_BAD_VALUE. Out-f-range value passed to SetTimeout().
- (Note that it's not considered an error if a timeout expires.)

SetBlocking() see Read()

SetDataBits() , SetStopBits() , SetParityMode() , DataBits() , StopBits() , ParityMode() , data_bits , stop_bits , parity_mode

void **SetDataBits(**data_bits *count***)**
void **SetStopBits(**stop_bits *count***)**

void **SetParityMode(**parity_mode *mode***)**
data_bits **DataBits(**void**)**
stop_bits **StopBits(**void**)**
parity_mode **ParityMode(**void**)**
typedef enum { **B_DATA_BITS_7**, **B_DATA_BITS_8** } **data_bits**
typedef enum { **B_STOP_BITS_1**, **B_STOP_BITS_2** } **stop_bits**
typedef enum { **B_EVEN_PARITY**, **B_ODD_PARITY**, **B_NO_PARITY** } **parity_mode**

These functions set and return characteristics of the serial unit used to send and receive data.

- **SetDataBits()** sets the number of bits of data in each unit; the default is **B_DATA_BITS_8**.
- **SetStopBits()** sets the number of stop bits in each unit; the default is **B_STOP_BITS_2**.
- **SetParityMode()** sets whether the serial unit contains a parity bit and, if so, the type of parity used; the default is **B_NO_PARITY**.

SetDataRate() , DataRate() , data_rate

status_t **SetDataRate(**data_rate *bitsPerSecond***)**
data_rate **DataRate(**void**)**
typedef enum { **B_0_BPS**, **B_50_BPS**, **B_75_BPS**, **B_110_BPS**, **B_134_BPS**, **B_150_BPS**, **B_200_BPS**, **B_30** **B_600_BPS**, **B_1200_BPS**, **B_1800_BPS**, **B_2400_BPS**, **B_4800_BPS**, **B_9600_BPS**, **B_19200_BPS**, **B_3125(** **B_38400_BPS**, **B_57600_BPS**, **B_115200_BPS**, **B_230400_BPS** } **data_rate**

These functions set and return the rate (in bits per second) at which data is both transmitted and received.

The default data rate is **B_19200_BPS**. If the rate is set to 0 (**B_0_BPS**), data will be sent and received at an indeterminate number of bits per second.

RETURN CODES

SetDataRate() returns...

- **B_OK. The rate was successfully set.**
- **B_NO_INIT**. The BSerialPort object doesn't have a port open.
- **B_ERROR.** The data rate couldn't be set for any other reason.

SetDTR()

status_t **SetDTR(**bool *pinAsserted***)**

Asserts the Data Terminal Ready (DTR) pin if the *pinAsserted* flag is `true`, and de-asserts it if the flag is `false`. The function should always returns **B_OK**.

SetFlowControl() , FlowControl()

void **SetFlowControl(**uint32 *mask*)
uint32 **FlowControl(**void)

These functions set and return the type of flow control the driver should use. There are four possibilities:

Control Code	Meaning
B_SOFTWARE_CONTROL	Control is maintained through XON and XOFF characters inserted into the data stream.
B_HARDWARE_CONTROL	Control is maintained through the Clear to Send (CTS) and Request to Send (RTS) pins.
B_SOFTWARE_CONTROL + **B_HARDWARE_CONTROL**	Both of the above.
0 (zero)	No control.

SetFlowControl() should be called before a specific serial port is opened. You can't change the type of flow control the driver uses in midstream.

SetParityMode() see SetDataBits()

SetRTS()

status_t **SetRTS(**bool *pinAsserted*)

Asserts the Request to Send (RTS) pin if the *pinAsserted* flag is `true`, and de-asserts it if the flag is `false`. The function always returns **B_OK**.

SetStopBits() see SetDataBits()
SetTimeout() see Read()
StopBits() see SetDataBits()

WaitForInput()

ssize_t **WaitForInput(**void)

Waits for input data to arrive at the serial port and returns the number of bytes available to be read. If data is already waiting, the function returns immediately.

This function doesn't respect the flag set by `SetBlocking()`; it blocks even if blocking is turned off for the `Read()` function. However, it does respect the timeout set by `SetTimeout()`. If the timeout expires before input data arrives at the serial port, it returns 0.

Write()

```
ssize_t Write(const void *data, size_t numBytes)
```

Writes up to *numBytes* of *data* to the serial port's output buffer. This function will be successful in writing the data only if the BSerialPort object has a port open. The output buffer holds a maximum of 512 bytes (1024 on Mac hardware).

RETURN CODES

non-negative integer. Success; the value is the number of bytes that were written.

- `B_INTERRUPTED`. The operation was interrupted by a signal.
- `B_FILE_ERROR`. The BSerialPort doesn't have a port open.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSerialPort Summary

The Device Kit Table of Contents The Device Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Device Kit

The Device Kit contains software for controlling hardware that's connected to the computer.

- A BSerialPort object represents a RS–232 serial port.
- A BJoystick object represents a joystick connection.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BJoystick Summary

The Device Kit Table of Contents The Device Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Drawing

This section discusses the framework in which BViews draw. Detailed descriptions of the functions mentioned here can be found in the BView and BWindow class descriptions.

View Geometry

Ech view is assigned a coordinate system of its own. By default, the coordinate origin—(0.0, 0.0)—is located at the left top corner of the view rectangle. The *x*-axis extends to the right and the *y*-axis extends downward; coordinate units count screen pixels. (For a detailed discussion of the coordinate systems assumed by the Interface Kit, see "The Coordinate Space" on page11.)

When a view is added as a child of another view, it's located within the coordinate system of its parent. A child is considered part of the contents of the parent view. If the parent moves, the child moves with it; if the parent view scrolls its contents, the child view is shifted along with everything else in the view.

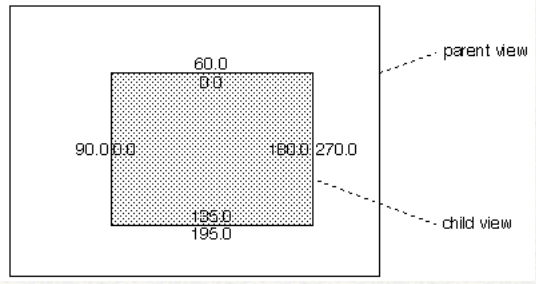
Since each view retains its own internal coordinate system no matter who its parent is, where it's located within the parent, or where the parent is located, a BView's drawing and message-handling code doesn't need to be concerned about anything exterior to itself. To do its work, a BView need look no farther than the boundaries of its own view rectangle.

Frame and Bounds Rectangles

Although a BView doesn't have to look outside its own boundaries, it does have to know where those boundaries are. It can get this information in two forms:

- Since a view is located within the coordinate system of its parent, the view rectangle is initially defined in terms of the parent's coordinates. This defining rectangle for a view is known as its *frame rectangle*. (See the BView constructor and the **`Frame()`** function.)
- When translated from the parent's coordinates to the internal coordinates of the view itself, the same rectangle is known as the *bounds rectangle*. (See the **`Bounds()`** function.)

The following illustration shows a child view 180.0 units wide and 135.0 units high. When viewed from the outside, from the perspective of its parent's coordinate system, it has a frame rectangle with left, top, right, and bottom coordinates at 90.0, 60.0, 270.0, and 195.0, respectively. But when viewed from the inside, in the view's own coordinate system, it has a bounds rectangle with coordinates at 0.0, 0.0, 180.0, and 135.0:



When a view moves to a new location in its parent, its frame rectangle changes but not its bounds rectangle. When a view scrolls its contents, its bounds rectangle changes, but not its

frame. The frame rectangle positions the view in the world outside; the bounds rectangle positions the contents inside the view.

Since a BView does its work in its own coordinate system, it refers to the bounds rectangle more often than to the frame rectangle.

Nonfractional Coordinates

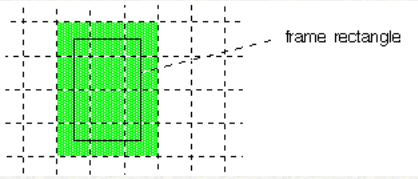
Because views are areas within windows and windows are displayed on-screen, the edges of a view must line up on rows and columns of screen pixels. It's easy to achieve this result, since coordinate units correspond to screen pixels; one unit is the distance from the center of a pixel to the center of an adjacent pixel. Therefore, all you must do is define the view rectangle with nonfractional coordinates, as in the illustration above.

Pixel and Coordinate Dimensions

It was mentioned that the child view in the illustration above is 180.0 coordinate units wide and 135.0 units high. However, this view actually covers 181 pixel columns and 136 pixel rows on-screen.

Two facts conspire to determine this result. First, as stated, coordinate units correspond to screen pixels. Second, the coordinate axes don't lie between pixels but right in the middle of them. The *x*-axis splits a row of pixels and the *y*-axis runs down the center of a column of pixels; the coordinate origin where they meet is at the very center of a pixel. Therefore, a view rectangle covers one more pixel in each direction than its coordinate dimensions would indicate.

Imagine an implausibly tiny frame rectangle like the one in the following diagram:



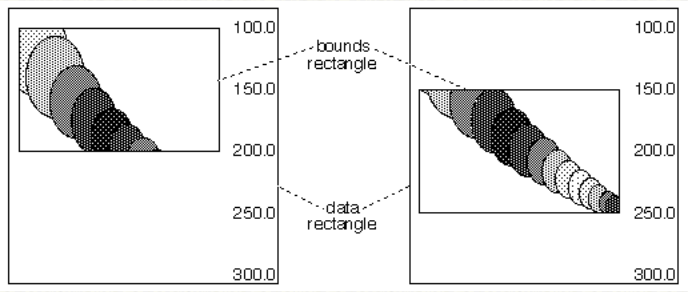
Since the pixels on the edges of this 2.0 * 3.0 rectangle are treated as being inside the rectangle, the view covers a 3 pixel * 4 pixel area.

This fact is important when laying out views and drawing in the bounds rectangle (or drawing any rectangle, for that matter). However, when discussing view coordinates and rectangles in general, it's easier and more accurate to speak in terms of coordinate values, not pixels—and that's the practice in this chapter. However, see "Mapping Coordinates to Pixels" on page14 for more on how coordinate values relate to pixels.

Scrolling

A BView scrolls its contents by shifting coordinate values within the view rectangle—that is, by altering the bounds rectangle. If, for example, the top of a view's bounds rectangle is at 100.0 and its bottom is at 200.0, scrolling downward 50.0 units would put the top at 150.0 and the

bottom at 250.0. Contents of the view with *y*-coordinate values of 150.0 to 200.0, originally displayed in the bottom half of the view, would be shifted to the top half. Contents with *y*-coordinate values from 200.0 to 250.0, previously unseen, would become visible at the bottom of the view, as shown in the following illustration:



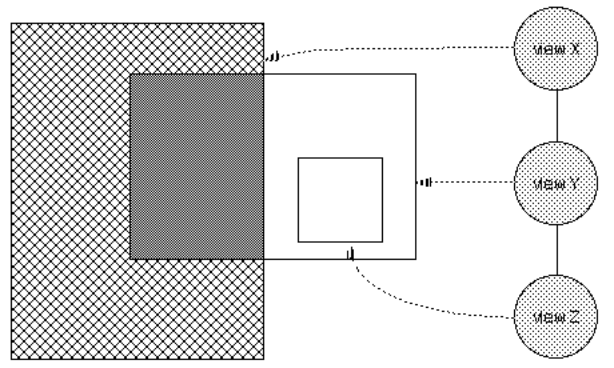
Scrolling doesn't move the view—it doesn't alter the frame rectangle—it moves only what's displayed inside the view. In the illustration above, a "data rectangle" encloses everything the BView is capable of drawing. For example, if the view is able to display an entire book, the data rectangle would be large enough to enclose all the lines and pages of the book laid end to end. However, since a BView can draw only within its bounds rectangle, everything in the data rectangle with coordinates that fall outside the bounds rectangle would be invisible. To make unseen data visible, the bounds rectangle must change the coordinates that it encompasses. Scrolling can be thought of as sliding the view's bounds rectangle to a new position on its data rectangle, as is shown in the illustration above. However, as it appears to the user, it's moving the data rectangle under the bounds rectangle. The view doesn't move; the data does.

The Clipping Region

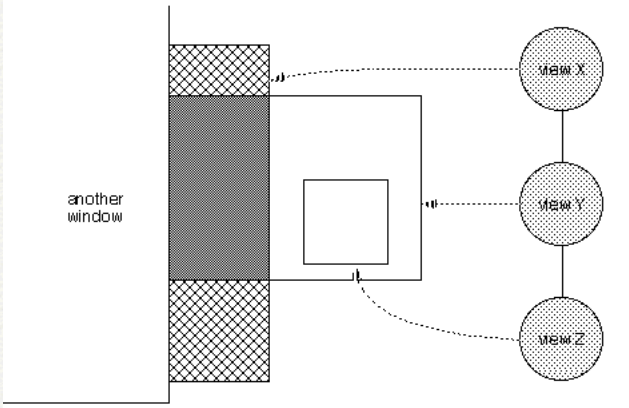
The Application Server clips the images that a BView produces to the region where it's permitted to draw.

This region is never any larger than the view's bounds rectangle; a view cannot draw outside its bounds. Furthermore, since a child is considered part of its parent, a view can't draw outside the bounds rectangle of its parent either—or, for that matter, outside the bounds rectangle of any ancestor view. In addition, since child views draw after, and therefore logically in front of, their parents, a view concedes some of its territory to its children.

Thus, the *visible region* of a view is the part of its bounds rectangle that's inside the bounds rectangles of all its ancestors, minus the frame rectangles of its children. This is illustrated in the following figure. It shows a hierarchy of three views—X, Y, and Z. The area filled with a crosshatch pattern is the visible region of view X; it omits the area occupied by its child, view Y. The visible region of view Y is colored dark gray; it omits the part of the view that lies outside its parent. View Z has no visible region, for it lies outside the bounds rectangle of its ancestor, view X:



The visible region of a view might be further restricted if its window is obscured by another window or if the window it's in lies partially off-screen. The visible region includes only those areas that are actually visible to the user. For example, if the three views in the previous illustration were in a window that was partially blocked by another window, their visible regions might be considerably smaller. This is shown in the next figure:



Note that in this case, view X has a discontinuous visible region.

The Application Server clips the drawing that a view does to a region that's never any larger than the visible region. On occasion, it may be smaller. For the sake of efficiency, while a view is being automatically updated, the *clipping region* excludes portions of the visible region that don't need to be redrawn:

- When a view is scrolled, the Application Server may be able to shift some of its contents from one portion of the visible region to another. The clipping region excludes any part of the visible region that the server was able to update on its own; it includes only the part where the BView must produce images that were not previously visible.
- If a view is resized larger, the clipping region may include only the new areas that were added to the visible region. (But see the *flags* argument for the BView constructor.)
- If only part of a view is invalidated (by the `Invalidate()` function), the clipping region is the intersection of the visible region and the invalid rectangle.

An application can also limit the clipping region for a view by passing a BRegion object to **ConstrainClippingRegion()**. The clipping region won't include any areas that aren't in the region passed. The Application Server calculates the clipping region as it normally would but intersects it with the specified region.

You can obtain the current clipping region for a view by calling **GetClippingRegion()**. (See also the BRegion class description.)

The View Color

Every view has a basic, underlying color. It's the color that fills the view rectangle before the BView does any drawing. The Application Server paints the view with this color before any view-specific drawing functions are called. The user may catch a glimpse of the color when the view is first shown on-screen, when it's resized larger, and when it's erased in preparation for an update. It will also be seen wherever the BView fails to draw in the visible region.

In a sense, the view color is the canvas on which the BView draws. It doesn't enter into any of the object's drawing operations except to provide a background. Although it's one of the BView's graphics parameters, it's not one that any drawing functions refer to.

The default view color is white. You can assign a different color to a view by calling BView's **SetViewColor()** function. If you set the view color to **B_TRANSPARENT_COLOR**, the Application Server won't erase the view's clipping region before an update. This is appropriate only if the view erases itself by touching every pixel in the clipping region when it draws.

The Background Bitmap

Every view may additionally have a background bitmap. The Application Server draws this bitmap after it fills in the view color and before any view-specific drawing functions are called. The view color will be visible in regions the background bitmap doesn't cover.

A view begins life without a background bitmap. A background bitmap may be added with BView's **SetViewBitmap()** function and subsequently removed with **ClearViewBitmap()**.

The Mechanics of Drawing

Views draw through the following set of primitive functions:

- `DrawString()` draws a string of characters. **`DrawChar()`** is a variant of this function; it draws just a single character.
- `DrawPicture()` executes a set of recorded drawing instructions.
- `DrawBitmap()` produces an image from a bitmap.
- **`CopyBits()`** copies an image from one location to another.
- `FillEllipse()`, `FillRegion()`, and other `Fill...()` functions fill closed shapes.
- `StrokeLine()`, `StrokeArc()`, and other `Stroke...()` functions stroke lines along defined paths.
- **`BeginLineArray()`**, **`AddLine()`**, and **`EndLineArray()`** draw a set of straight lines, all of the same width, but possibly in different colors.

The way these functions work depends not only on the values that they're passed—the particular string, bitmap, arc, or ellipse that's to be drawn—but on previously set values in the BView's graphics environment.

Graphics Environment

Each BView object maintains its own graphics environment for drawing. The view color, coordinate system, and clipping region are fundamental parts of that environment, but not the only parts. It also includes a number of parameters that can be set and reset at will to affect the next image drawn. These parameters are:

- Font attributes that determine the appearance of text the BView draws. (See **`SetFont()`** and the BFont class.)
- Two pen parameters—a location and a size. The pen location determines where the next drawing will occur, and the pen size determines the thickness of stroked lines. (See **`MovePenBy()`** and **`SetPenSize()`**.)
- Two current colors—a *high color* and a *low color*—that can be used either alone or in combination to form a pattern or halftone. The high color is used for most drawing. The low color is sometimes set to the underlying view color so that it can be used to erase other drawing or, because it matches the view background, make it appear that drawing has not touched certain pixels.
- (The high and low colors roughly match what other systems call the fore and back, or foreground and background, colors. However, neither color truly represents the color of the foreground or background. The terminology *high* and *low* is meant to keep the sense of two opposing colors and to match how they're defined in a pattern. A pattern bit is turned on for the high color and turned off for the low color. See the **`SetHighColor()`** and **`SetLowColor()`** functions in the BView class description and the "Patterns" section on .)

- A drawing mode that determines how the next image is to be rendered. (See "Drawing Modes" on page19 and the **`SetDrawingMode()`** function.)

By default, a BView's graphics parameters are set to the following values:

Font	The system plain font (**`be_plain_font`**)
Pen position	(0.0, 0.0)
Pen size	1.0 coordinate units
High color	Black (red, green, and blue components all equal to 0)
Low color	White (red, green, and blue components all equal to 255)
Drawing mode	Copy mode (`B_OP_COPY`)
View color	White (red, green, and blue components all equal to 255)
Clipping region	The visible region of the view
Coordinate system	Origin at the left top corner of the bounds rectangle

However, as "Views and the Server" on page24 explains, these values have meaning only when the BView is assigned to a window.

The Pen

The pen is a fiction that encompasses two properties of a view's graphics environment: the current drawing location and the thickness of stroked lines.

The pen location determines where the next image will be drawn—but only if another location isn't explicitly passed to the drawing function. Some drawing functions alter the pen location—as if the pen actually moves as it does the drawing—but usually it's set by calling **`MovePenBy()`** or **`MovePenTo()`**.

The pen that draws lines (through the various `Stroke...()` functions) has a malleable tip that can be made broader or narrower by calling the **`SetPenSize()`** function. The larger the pen size, the thicker the line that it draws.

The pen size is expressed in coordinate units, which must be translated to a particular number of pixels for the display device. This is done by scaling the pen size to a device-specific value and rounding to the closest integer. For example, pen sizes of 2.6 and 3.3 would both translate to 3 pixels on-screen, but to 7 and 10 pixels respectively on a 300dpi printer.

The size is never rounded to 0; no matter how small the pen may be, the line never disappears. If the pen size is set to 0.0, the line will be as thin as possible—it will be drawn using the fewest possible pixels on the display device. (In other words, it will be rounded to 1 for all devices.)

If the pen size translates to a tip that's broader than 1 pixel, the line is drawn with the tip centered on the path of the line. Roughly the same number of pixels are colored on both sides of the path.

A later section, "Picking Pixels to Stroke and Fill" on page15, illustrates how pens of different sizes choose the pixels to be colored.

Colors

The high and low colors are specified as `rgb_color` values—full 32-bit values with separate red, green, and blue color components, plus an alpha component for transparency. Although there may sometimes be limitations on the colors that can be rendered on-screen, there are no restrictions on the colors that can be specified.

Color Spaces

The way colors are specified for a bitmap depends on the color space in which they're interpreted. The color space determines the *depth* of the bitmap data (how many bits of information are stored for each pixel), the *interpretation* of the data (whether it represents shades of gray or true colors, whether it's segmented into color components, what the components are, and so on), and the *arrangement* of components within the data (whether big-endian or little-endian). These six basic color spaces are recognized:

`B_GRAY1`	One bit of data per pixel, where 1 is black and 0 is white.
`B_GRAY8`	Eight bits of data per pixel, where a value of 255 is black and 0 is white.
`B_CMAP8`	Eight bits of data per pixel, interpreted as an index into a list of 256 colors. The list is part of the system color map and is the same for all applications.
`B_RGB15`	Three components of data per pixel—blue, green, and red, in that order—with 5 bits each for red, green, and blue. The first bit of the pixel data is not associated with any color component.
`B_RGBA15`	Four components of data per pixel—blue, green, red, and alpha, in that order—with 5 bits each for red, green, and blue, and 1 bit for alpha.
`B_RGB16`	Three components of data per pixel—blue, green, and red, in that order—with 5 bits each for red and blue, and 6 bits for green.
`B_RGB32`	Four components of data per pixel—blue, green, red, and alpha, in that order—with 8 bits per component. A component value of 255 yields the maximum amount of red, green, or blue, and a value of 0 indicates the absence of that color.

The components in the `B_RGB32`, `B_RGB16`, and `B_RGB15` color spaces are meshed rather than separated into distinct planes; all four components are specified for the first pixel before the four components for the second pixel, and so on. The order of bytes for these two types is little-endian, which means that for `B_RGB32` data, the component bytes appear in the order blue, green, red, and alpha.

Counterpart color spaces are defined for big-endian data. `B_RGB32_BIG`, `B_RGB16_BIG`, `B_RGBA15_BIG`, and `B_RGB15_BIG`, are equivalent to `B_RGB32`, `B_RGB16`, `B_RGBA15`, and `B_RGB15`—except for the order of bytes. The Be operating system retains data in the little-endian formats; the big-endian color spaces are defined only to label noncompatible data that the system must convert and to allow drivers to communicate precise formats to the operating system.

Alpha should be 0 for 100% transparent, or 255 (for 8-bit alpha channels) for 100% opaque. For 1-bit alpha channels, a value of 1 indicates opaque.

The Screen

The screen can be configured to display colors in either the **B_CMAP8**, **B_RGB15**, or **B_RGB32** color spaces. When it's in the **B_CMAP8** or **B_RGB15** color spaces, specified **rgb_color**s are displayed as the closest available color. (See the BBitmap and BScreen classes.)

Patterns

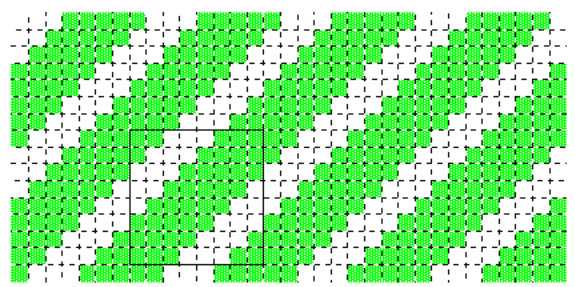
Most functions that stroke a line or fill a closed shape don't draw directly in either the high or the low color. Rather they take a *pattern*, an arrangement of one or both colors that's repeated over the entire surface being drawn. A pattern might consist of just the high color, just the low color, or some combination of the two.

By combining the low color with the high color, patterns can produce dithered colors that lie somewhere between two hues in the **B_CMAP8** color space. Patterns also permit drawing with less than the solid high color (for intermittent or broken lines, for example) and can take advantage of drawing modes that treat the low color as if it were transparent, as discussed in the next section.

A pattern is defined as an 8-pixel by 8-pixel square. The **pattern** type is 8 bytes long, with 1 byte per row and 1 bit per pixel. Rows are specified from top to bottom and pixels from left to right. Bits marked 1 designate the high color; those marked 0 designate the low color. For example, a pattern of wide diagonal stripes could be defined as follows:

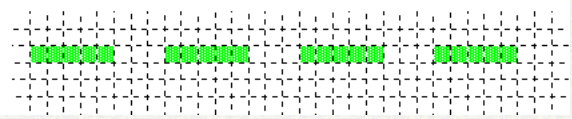
```
pattern stripes = { 0xc7, 0x8f, 0x1f, 0x3e,
                    0x7c, 0xf8, 0xf1, 0xe3 };
```

Patterns repeat themselves across the screen, like tiles laid side by side. The pattern defined above looks like this:



The dotted lines in this illustration show the separation of the screen into pixels. The thicker black line outlines one 8 * 8 square that the pattern defines.

The outline of the shape being filled or the width of the line being stroked determines where the pattern is revealed. It's as if the screen was covered with the pattern just below the surface, and stroking or filling allowed some of it to show through. For example, stroking a 1-pixel wide horizontal path in the pattern illustrated above would result in a dotted line, with the dashes (in the high color) slightly longer than the spaces between (in the low color):



When stroking a line or filling a shape, the pattern serves as the source image for the current drawing mode, as explained in "Drawing Modes" next. The nature of the mode determines how the pattern interacts with the destination image, the image already in place.

The Interface Kit defines three patterns:

B_SOLID_HIGH	Consists only of the high color.
B_SOLID_LOW	Has only the low color.
B_MIXED_COLORS	Mixes the two colors evenly, like the pattern on a checkerboard.

B_SOLID_HIGH is the default pattern for all drawing functions. Applications can define as many other patterns as they need.

Drawing Modes

When a BView draws, it in effect transfers an image to a target location somewhere in the view rectangle. The drawing mode determines how the image being transferred interacts with the image already in place at that location. The image being transferred is known as the *source image*; it might be a bitmap or a pattern of some kind. The image already in place is known as the *destination image*.

In the simplest and most straightforward kind of drawing, the source image is simply painted over the destination; the source replaces the destination. However, there are other possibilities. There are ten different drawing modes—ten distinct ways of combining the source and destination images. The modes are designated by **drawing_mode** constants that can be passed to **SetDrawingMode()**:

B_OP_COPY	**B_OP_ADD**
B_OP_OVER	**B_OP_SUBTRACT**
B_OP_ERASE	**B_OP_BLEND**
B_OP_INVERT	**B_OP_MIN**
B_OP_SELECT	**B_OP_MAX**

B_OP_COPY is the default mode and the simplest. It transfers the source image to the destination, replacing whatever was there before. The destination is ignored.

In the other modes, however, some of the destination might be preserved, or the source and destination might be combined to form a result that's different from either of them. For these modes, it's convenient to think of the source image as an image that exists somewhere independent of the destination location, even though it's not actually visible. It's the image that would be rendered at the destination in **B_OP_COPY** mode.

Bitmaps and Patterns

The modes work for all BView drawing functions—including those that stroke lines and fill shapes, those that draw characters, and those that image bitmaps. The way they work depends foremost on the nature of the source image—whether it's a *pattern* or a *bitmap*. For the `Fill...()` and `Stroke...()` functions, the source image is a pattern that has the same shape as the area being filled or the area the pen touches as it strokes a line. For **`DrawBitmap()`**, the source image is a rectangular bitmap.

In a sense, a pattern is simply a bitmap that's one bit deep. It's a bitmap consisting of two colors, one which maps to the current high color and another that maps to the current low color. As we shall see later, a **`B_MONOCHROME_1_BIT`** bitmap acts just like a pattern. However, patterns and bitmaps generally behave differently:

- Only a source pattern has designated high and low colors. Even if a source bitmap has colors that match the current high and low colors, they're not handled like the colors in a pattern; they're treated just like any other color in the bitmap.
- On the other hand, only a source bitmap can have transparent pixels. In the **`B_CMAP8`** color space, a pixel is made transparent by assigning it the **`B_TRANSPARENT_MAGIC_CMAP8`** value. In the **`B_RGB32`** color space, a pixel assigned the **`B_TRANSPARENT_MAGIC_RGBA32`** value is considered transparent. These values have meaning only for source bitmaps, not for source patterns. If the current high or low color in a pattern happens to have a transparent value, it's still treated as the high or low color, not like transparency in a bitmap. For big endian color spaces, the transparency color should be byte-swapped from its little endian counterpart.

B_TRANSPARENT_MAGIC_CMAP8	8-bit indexed color transparent pixel.
B_TRANSPARENT_MAGIC_RGBA15	15-bit transparent pixel.
B_TRANSPARENT_MAGIC_RGBA15 _BIG	15-bit transparent pixel, big-endian.
B_TRANSPARENT_MAGIC_RGBA32	32-bit transparent pixel.
B_TRANSPARENT_MAGIC_RGBA32 _BIG	32-bit transparent pixel, big-endian.

Drawing Modes and Color Spaces

The way the drawing modes work also depends on the color space of the source image and the color space of the destination. The following discussion concentrates on drawing where the source and destination both contain colors. This is the most common case, and also the one that's most general.

The source and destination images can have different color spaces. For example, a source bitmap might be defined in the **`B_CMAP8`** space while the destination is displayed in the full color **`B_RGB32`** color space. The drawing operation merely combines the colors in the two images in some way. It doesn't transfer the color space of the source image to the destination. The image that results from the drawing operation will always be in the color space of the destination image.

Mode Definitions

When applied to colors, the ten drawing modes fall naturally into four groups:

- The `B_OP_COPY` mode, which copies the source image to the destination.
- The `B_OP_OVER`, `B_OP_ERASE`, `B_OP_INVERT`, and `B_OP_SELECT` modes, which—despite their differences—all treat the low color in a pattern as if it were transparent.
- The `B_OP_ADD`, `B_OP_SUBTRACT`, and `B_OP_BLEND` modes, which combine colors in the source and destination images.
- The `B_OP_MIN` and `B_OP_MAX` modes, which choose between the source and destination colors.

The following paragraphs describe each of these groups in turn.

Copy mode

In `B_OP_COPY` mode, the source image replaces the destination. This is the default drawing mode and the one most commonly used. Because this mode doesn't have to test for particular color values in the source image, look at the colors in the destination, or compute colors in the result, it's also the fastest of the modes.

If the source image contains transparent pixels, their transparency will be retained in the result; the transparent value is copied just like any other color. However, the appearance of a transparent pixel when shown on-screen is indeterminate. If a source image has transparent portions, it's best to transfer it to the screen in `B_OP_OVER` or another mode. In all modes other than `B_OP_COPY`, a transparent pixel in a source bitmap preserves the color of the corresponding destination pixel.

Transparency modes

Four drawing modes—`B_OP_OVER`, `B_OP_ERASE`, `B_OP_INVERT`, and `B_OP_SELECT`—are designed specifically to make use of transparency in the source image; they're able to preserve some of the destination image. In these modes (and only these modes) the low color in a source pattern acts just like transparency in a source bitmap.

Each of these modes has a different effect on the destination image—but only in those places where the source image is not transparent. One of the modes, `B_OP_OVER`, transfers some of the source image to the destination. The other three modes play with the destination in some way—erase it, invert it, or select colors in it—without regard to the source image. For these modes, the only thing that matters about the source image is where it's transparent and where it's not. Each of the four modes is described below:

- The `B_OP_OVER` mode places the source image "over" the destination; the source provides the foreground and the destination the background. In this mode, the source image replaces the destination image (just as in the `B_OP_COPY` mode)—except where a source bitmap has transparent pixels and a source pattern has the low color. Transparency in a bitmap and the low color in a pattern retain the destination image in the result.
- By masking out the unwanted parts of a rectangular bitmap with transparent pixels, this mode can place an irregularly shaped source image in front of a background image. Transparency in the source foreground lets the destination background show through.

The versatility of B_OP_OVER makes it the second most commonly used mode, after B_OP_COPY.

- The B_OP_ERASE mode doesn't draw the source image at all. Instead, it erases the destination image. Like B_OP_OVER, it preserves the destination image wherever a source bitmap is transparent or a source pattern has the low color. But everywhere else—where the source bitmap isn't transparent and the source pattern has the high color—it removes the destination image, replacing it with the low color.
- Although this mode can be used for selective erasing, it's simpler to erase by filling an area with the **B_SOLID_LOW** pattern in B_OP_COPY mode.
- The B_OP_INVERT mode, like B_OP_ERASE, doesn't draw the source image. Instead, it inverts the colors in the destination image. As in the case of the B_OP_OVER and B_OP_ERASE modes, where a source bitmap is transparent or a source pattern has the low color, the destination image remains unchanged in the result. Everywhere else, the color of the destination image is inverted.
- The inversion of an rgb_color is the complement of its color components. For example, the inversion of a red value of 58 would be 197 (255–58).
- The B_OP_SELECT mode also doesn't draw the source image. It replaces the high color in the destination image with the low color and the low color with the high color. As for the other modes in this group, where a source bitmap is transparent or a source pattern has the low color, the destination image remains unchanged in the result. Everywhere else, the high and low colors are switched.
- This is similar to the B_OP_INVERT mode, except that B_OP_SELECT affects at most only two colors in the destination image. The destination is preserved not only where the source is transparent, but also where its colors don't match the current high and low colors.

These four modes also work for monochrome images. If the source image is monochrome, the distinction between source bitmaps and source patterns breaks down. Two rules apply:

- If the source image is a monochrome bitmap, it acts just like a pattern. A value of 1 in the bitmap designates the current high color, and a value of 0 designates the current low color. Thus, 0, rather than B_TRANSPARENT_MAGIC_*, becomes the transparent value.
- If the source and destination are both monochrome, the high color is necessarily black (1), and the low color is necessarily white (0)—but otherwise the drawing modes work as described. With the possible colors this severely restricted, the three modes are reduced to boolean operations: B_OP_OVER is the same as a logical *OR*, B_OP_INVERT and B_OP_SELECT are the same as logical *exclusive OR*, and B_OP_ERASE is the same as an inversion of logical *AND*.

Blending modes

Three drawing modes—B_OP_ADD, B_OP_SUBTRACT, and B_OP_BLEND—combine the source and destination images, pixel by pixel, and color component by color component. As in most of the other modes, transparency in a source bitmap preserves the destination image in the

result. Elsewhere, the result is a combination of the source and destination. The high and low colors of a source pattern aren't treated in any special way; they're handled just like other colors.

- `B_OP_ADD` adds each component of the source color to the corresponding component of the destination color, with a component value of 255 as the limit. Colors become brighter, closer to white.
- By adding a uniform gray to each pixel in the destination, for example, the whole destination image can be brightened by a constant amount.
- `B_OP_SUBTRACT` subtracts each component of the source color from the corresponding component of the destination color, with a component value of 0 as the limit. Colors become darker, closer to black.
- For example, by subtracting a uniform amount from the red component of each pixel in the destination, the whole image can be made less red.
- `B_OP_BLEND` averages each component of the source and destination colors (adds the source and destination components and divides by 2). The two images are merged into one.

These modes work only for color images, not for monochrome ones. If the source or destination is specified in the `B_CMAP8` color space, the color will be expanded to a full `B_RGB32` value to compute the result; the result is then contracted to the closest color in the `B_CMAP8` color space.

Selection modes

Two drawing modes—`B_OP_MAX` and `B_OP_MIN`—compare each pixel in the source image to the corresponding pixel in the destination image and select one to keep in the result. If the source pixel is transparent, both modes select the destination pixel. Otherwise, `B_OP_MIN` selects the darker of the two colors and `B_OP_MAX` selects the brighter of the two. If the source image is a uniform shade of gray, for example, `B_OP_MAX` would substitute that shade for every pixel in the destination image that was darker than the gray.

Like the blending modes, `B_OP_MIN` and `B_OP_MAX` work only for color images.

Views and the Server

Windows lead a dual life—as on-screen entities provided by the Application Server and as BWindow objects in the application. BViews have a similar dual existence—each BView object has a shadow counterpart in the server. The server knows the view's location, its place in the window's hierarchy, its visible area, and the current state of its graphics parameters. Because it has this information, the server can more efficiently associate a user action with a particular view and interpret the BView's drawing instructions.

BWindows become known to the Application Server when they're constructed; creating a BWindow object causes the server to produce the window that the user will eventually see on-screen. A BView, on the other hand, has no effect on the server when it's constructed. It becomes known to the server only when it's attached to a BWindow. The server must look through the application's windows to see what views it has.

A BView that's not attached to a window therefore lacks a counterpart in the server. This restricts what some functions can do. Three groups of functions are affected:

- Drawing functions—**`DrawBitmap()`**, **`FillRect()`**, **`StrokeLine()`**, and so on—don't work for unattached views. A BView can't draw unless it's in a window.
- The scrolling functions—**`ScrollTo()`** and **`ScrollBy()`**—require the BView to be in a window. Manipulations of a view's coordinate system are carried out in its server counterpart.
- Functions that indirectly depend on a BView's graphics parameters—such as **`GetMouse()`**, which reports the cursor location in the BView's coordinates—also require the BView to belong to a window. These functions need information that an unattached BView can't provide.

However, the functions that set and return graphics parameters—such as **`SetFont()`**, **`SetDrawingMode()`**, **`PenLocation()`**, and **`SetHighColor()`**—are not restricted. A view's graphic state is kept within the server (where it's needed to carry out drawing instructions), but also cached by the BView. Therefore, it's possible to assign a value to a graphics parameter before the server knows about the view. The value is simply cached until the view becomes part of a window's view hierarchy; the BView then hands it to the server. The server and the client-side cache are always kept in synch.

Attaching to a Window

Although you can set a BView's graphics parameters before it belongs to a window and has a counterpart in the Application Server, some of its initialization may need to wait until the BView receives an **`AttachedToWindow()`** notification informing it that it has been added to a window's view hierarchy. For example, if a view adopts the background color of its parent, it can only set the view color in **`AttachedToWindow()`**:

```
void MyView::AttachedToWindow(void)
{
   if ( Parent() )
```

```
        SetViewColor(Parent()->ViewColor());
    . . .
}
```

AttachedToWindow() is called for each view that's added to a window, beginning with the root view being attached, followed by each of its children, and so on down the hierarchy. After all views have been notified with an **AttachedToWindow()** function call, they each get an **AllAttached()** notification, but in the reverse order. A parent view that must adjust itself to calculations made by a child view when it's attached to a window can wait until **AllAttached()** to do the work.

These two function calls are matched by another pair—**DetachedFromWindow()** and **AllDetached()**—which notify BViews that they're about to be removed from the window.

Preparing to Draw

A BView doesn't have to draw anything within its frame rectangle—it can just be a container for other BViews that do draw there. However, most views that you implement will draw. And most views draw by implementing the **Draw()** function. This function is called upon to present the view on-screen (or, when printing, on a page). It's implemented using the primitive drawing functions listed above. If your BView includes a **Draw()** implementation, you must include **B_WILL_DRAW** in the BView's flags upon construction.

The Update Mechanism

The Application Server sends a message to a BWindow whenever any of the views within the window need to be updated. The BWindow then calls the **Draw()** function of each out-of-date BView so that it can redraw the contents of its on-screen display.

Update messages can arrive at any time. A BWindow receives one when the window is first placed on screen, or is shown after being hidden

- The window is first placed on-screen, or is shown again after having been hidden.
- Any part of the window becomes visible after being obscured.
- The views in the window are rearranged—for example, if a view is resized or a child is removed from the hierarchy.
- Something happens to alter what a particular view displays. For example, if the contents of a view are scrolled, the BView must draw any new images that scrolling makes visible. If one of its children moves, it must fill in the area the child view vacated.
- The application forces an update by "invalidating" a view, or a portion of a view.

As update messages arrive, they jump to the head of the BWindow's message queue.

Forcing an Update

When a user action or a BView function alters a view—if the view is scrolled, for example—an update message is sent and the BView's **Draw()** function is automatically called. But if the **Draw()** function depends on some other state that's defined by your application, you need to tell the Application Server that your BView needs an update message. You do this by invoking BView's **Invalidate()** function.

For example, let's say your "D2DView" subclass connects some dots by reading points out of a BList and drawing lines between them:

```
void D2DView::Draw(BRect update)
{
   int32 i=1;
   if (dotList->CountItems() > 1) {
      MovePenTo(*(BPoint *)dotList->ItemAt(0);
      while (i < dotList->CountItems())
         StrokeLine(*(BPoint *)dotList->ItemAt(i++));
   }
}
```

Each "dot specification" is added through DotView's **AddDot()** function. **AddDot()** includes a call to **Invalidate()** in order to force the view to be redrawn:

```
void D2DView::AddDot(BPoint p)
{
   dotList->AddItem(new BPoint(p));
   Invalidate();
}
```

Forcing an Update while Responding to an Event

The hook functions that respond to user events—**KeyDown()**, **MouseDown()**, and so on—are executed in the same thread that receives window update messages. If you do something in your implementation of one of these hook functions that causes an update, the update message won't be processed until your hook function exits. If your hook function does a lot of processing, your interface can become unresponsive.

To get around this problem, call BWindow's **UpdateIfNeeded()** function from within your hook functions' implementations. **UpdateIfNeeded()** forces any pending update messages to be processed immediately. The function doesn't return until the "dirty" views are all done re-drawing. (Note that **UpdateIfNeeded()** *only* works from within the BWindow's message loop thread.)

For example, let's say you want to add a random dot (within a 100x100 square) everytime the user hits a key:

```
void D2DView::KeyDown(const char *bytes, int32 numBytes)
{
   AddDot(BPoint(rand()%100, rand()%100);
```

```
    Window()->UpdateIfNeeded();
    ...
}
```

Since **AddDot()** calls **Invalidate()**, we know that there's an update message pending. However, the update won't normally be processed until **KeyDown()** exits, so we call **UpdateIfNeeded()**, which sees the update and calls D2DView's **Draw()** function. When **Draw()** is finished, **UpdateIfNeeded()** returns and **KeyDown()** continues.

Erasing the Clipping Region

Just before sending an update message, the Application Server prepares the clipping region of each BView that is about to draw by erasing it to the view background color. Note that only the clipping region is erased, not the entire view, and perhaps not the entire area where the BView will, in fact, draw.

The server forgoes this step only if the BView's background color is set to **B_TRANSPARENT_COLOR**. See "The View Color" above.

Drawing during an Update

While drawing, a BView may set and reset its graphics parameters any number of times—for example, the pen position and high color might be repeatedly reset so that whatever is drawn next is in the right place and has the right color. These settings are temporary. When the update is over, all graphics parameters are reset to their initial values.

If, for example, `Draw()` sets the high color to a shade of light blue, as shown below,

```
SetHighColor(152, 203, 255);
```

it doesn't mean that the high color will be blue when `Draw()` is called next. If this line of code is executed during an update, light blue would remain the high color only until the update ends or **SetHighColor()** is called again, whichever comes first. When the update ends, the previous graphics state, including the previous high color, is restored.

Although you can change most graphics parameters during an update—move the pen around, reset the font, change the high color, and so on—the coordinate system can't be touched; a view can't be scrolled while it's being updated. If the view's coordinate system were to change, it would alter the current clipping region and confuse the update mechanism.

Drawing outside of an Update

Graphics parameters that are set outside the context of an update are not limited; they remain in effect until they're explicitly changed. For example, if application code calls **Draw()**, perhaps in response to an interface message, the parameter values that **Draw()** last sets would persist even after the function returns. They would become the default values for the view and would be assumed the next time **Draw()** is called.

Default graphics parameters are typically set as part of initializing the BView once it's attached to a window—in an `AttachedToWindow()` function. If you want a `Draw()` function to assume the values set by `AttachedToWindow()`, it's important to restore those values after any drawing the BView does that's not the result of an update. For example, if a BView invokes `SetHighColor()` while drawing in response to an interface message, it will need to restore the default high color when done.

If `Draw()` is called outside of an update, it can't assume that the clipping region will have been erased to the view color, nor can it assume that default graphics parameters will be restored when it's finished.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRegion

Derived from: *none*
Declared in: be/interface/Region.h
Library: libbe.so
more...

A BRegion object comprises a set of rectangles in a two-dimensional coordinate system. The rectangles needn't overlap; a single BRegion can contain a little rectangle way down there, a big one over yonder, and another that lives somewhere to the left of Baden Baden.

You define and modify the contents of a region through to `Set()`, `Include()`, `Exclude()`, and `IntersectWith()`. You can examine a region's rectangle's through `RectAt()`, but note that the object coalesces rectangles when it can, so the rectangles that you retrieve may not match the ones that you passed in.

The BRegion class is designed to be used with coordinates with integral values only; if any non-integral values are used in the BRegion, it functions in the class may behave unpredictably.

BRegion objects can be allocated on the stack.

Constructor and Destructor

BRegion()

```
BRegion(const BRegion& region)
BRegion(const BRect rect)
BRegion(void)
```

Initializes the BRegion object to be a copy of *region*, or to be a rectangular region containing the space indicated by *rect*. If no argument is supplied, the BRegion is empty and `Frame()` returns and invalid BRect.

~BRegion

```
virtual ~BRegion()
```

Frees the memory that was allocated to hold data describing the region.

Member Functions

Contains()

```
bool Contains(BPoint point) const
```

Returns `true` if *point* lies within the region, and `false` if not.

CountRects() see RectAt()

Exclude() see Set()

Frame()

```
BRect Frame(void) const
```

Returns the smallest rectangle that encloses all the rectangles in the region.

If the region is empty, the returned object will be invalid. Check the return thus:

```
rect = region.Frame();
if (rect.IsValid())
   /* okay */
else
   /* nope */
```

Include() see Set()

Intersects()

```
bool Intersects(BRect rect) const
```

Returns `true` if the BRegion has any area in common with *rect*, and `false` if not.

IntersectWith() see Set()

MakeEmpty()

```
void MakeEmpty(void)
```

Empties the BRegion and invalidates the object's `Frame()` rectangle.

OffsetBy()

void **OffsetBy(**int32 *horizontal*, int32 *vertical***)**

Offsets all rectangles in the region by *horizontal* and *vertical* deltas.

PrintToStream()

void **PrintToStream(**void**)** const

Prints (to standard out) each rectangle in the BRegion in the form:

```
"BRect(left, top, right, bottom)"
```

The first string that's printed is the object's frame rectangle. Each subsequent string describes a single rectangle in the BRegion.

RectAt() , CountRects()

BRect **RectAt(**int32 *index***)**
int32 **CountRects(**void**)**

`CountRects()` returns the total number of rectangles in the BRegion; `RectAt()` returns the rectangle at a particular *index*. The order of the rectangles is unimportant.

Set() , Include() , Exclude() , IntersectWith()

void **Set(**BRect *rect***)**
void **Include(**BRect *rect***)** void **Include(**const BRegion **region***region***)**
void **Exclude(**BRect *rect***)** void **Exclude(**const BRegion **region***region***)**
void **IntersectWith(**const BRegion **region***region***)**

`Set()` modifies the BRegion so that it only contains *rect*.

`Include()` modifies the region to include (copies of) the rectangles contained in *rect* or *region*.

`Exclude()` modifies the region to exclude all rectangles contained in *rect* or *region*..

`IntersectWith()` modifies the region so that it includes only those areas that it has in common with another *region*.

Operators

= (assignment)

```
BRegion& operator =(const BRegion&)
```

Sets the left region to be an independent copy of the right region:

```
BRegion left_region = right_region;
```

The Be Book The Interface Kit The Interface Kit Index

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRegion Function Summary

Function Summary

Function	Description
=	Sets one region to equal the other
Contains()	Tests whether a BPoint lies within the region
CountRects()	Returns the number of rectangles in the region
Exclude()	Removes rectangles from the region
Frame()	Returns the smallest rectangle enclosing all the rectangles in the region
Include()	Adds rectangles to the region
IntersectWith()	Modifies region to contain only rectangles in common with another region
Intersects()	Tests whether a BRect intersects the region
MakeEmpty()	Empties the BRegion
OffsetBy()	Offset all rectangles in the region by a given amount
PrintToStream()	Prints the coordinates of each rectangle in the region to standard output
RectAt()	Returns a specific rectangle in the region
BRegion()	Constructor
Set()	Empties the region and adds a single BRect to it

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Kernel Kit

Images

Declared in: be/kernel/image.h
more...

Memory , Initialization , and , Termination , Routines

#define **B_INIT_BEFORE_FUNCTION_NAME** "initialize_before"
#define **B_INIT_AFTER_FUNCTION_NAME** "initialize_after"
#define **B_TERM_BEFORE_FUNCTION_NAME** "terminate_before"
#define **B_TERM_AFTER_FUNCTION_NAME** "terminate_after"

x86 only!

The quoted strings are the names of the functions that are called just before and just after a chunk of memory is initialized, and just before and just after the memory is freed. You can provide your own initialization and termination routines by implementing functions with these names. The functions take no arguments and return void.

The "B_" symbols that represent the names are placeholders—they're not used anywhere in the BeOS.

---

get_cpuid(), cpuid_info

Declared in: be/kernel/OS.h

There have been changes to the (undocumented) **`get_cpuid()`** function and **`cpuid_info`** structure. However, the call and structure will be replaced by a higher-level API in a later release. If you use the current versions, you may have to rewrite your code when these versions are retired.

on_exit_thread(), exit_thread_func

Declared in: be/kernel/OS.h

```
status_t on_exit_thread(exit_thread_func func, void *data)
typedef void (*exit_thread_func) (void *);
```

on_exit_thread() lets you register an "exit thread" function that's called after the this thread has been stopped, but just before it exits entirely. The *data* argument is passed to *func* when the latter is called.

Note that *func* is registered as the exit function for the thread that calls on_exit_thread(), displacing any previously registered functions. The exit function is called in the exiting thread's context.

RETURN CODES

- B_OK. The function was installed properly.
- B_NO_MEMORY. Can't create the necessary structures to register the callback.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

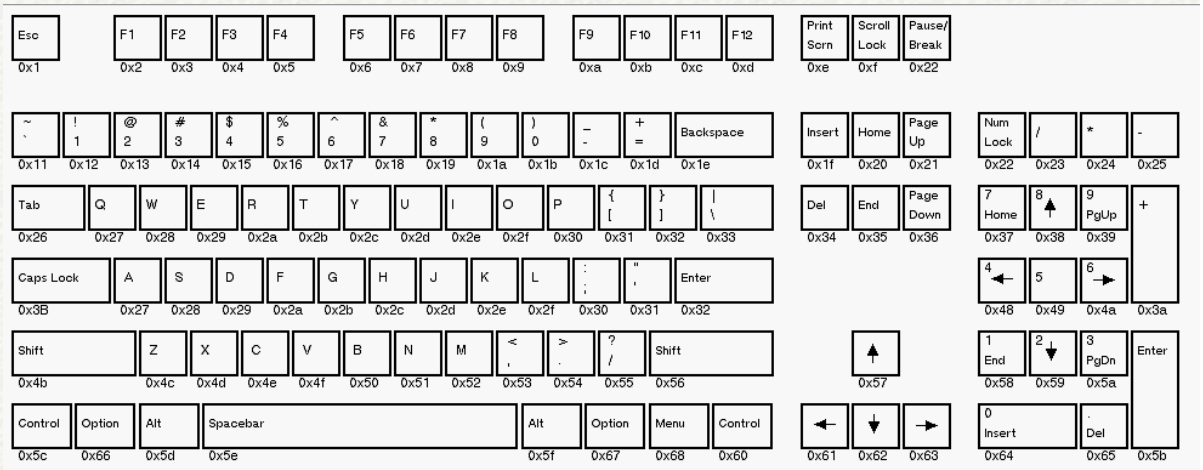
Copyright © 2000 Be, Inc. All rights reserved..

Key Codes

Each key on a computer's keyboard is assigned a numerical code which is reported as the "key" field in the **B_KEY_DOWN** or **B_UNMATCHED_KEY_DOWN** message. Likewise, the "key" field of the **B_KEY_UP** or **B_UNMATCHED_KEY_UP** message indicates which key was released.

The **Print Screen** key is trapped by BeOS; it doesn't generate a key down message, but it does generate a key up.

The following illustration shows the keycodes for most of the keys on standard keyboards.



Some keyboards vary slightly from this; however, even if keys are in different locations than those depicted here, they'll still have the same key code values.

Note that the **Option** key on Macintosh keyboards and the **Windows** key on many PC keyboards share the same keycode.

Some other keys that don't appear on this diagram include:

Key	Keycode
System Request	0x7E
Break	0x7F
Euro	0x69
Keypad =	0x6A
Mac Power Key	0x6B

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Key States

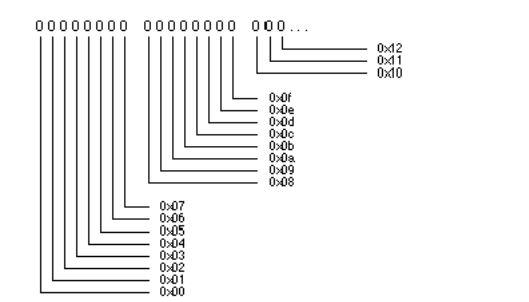
You can look at the state of all the keys on the keyboard at a given moment in time. This information is captured and reported in two ways:

- As the "states" field in every `B_KEY_DOWN` message, and
- As the `key_states` bitfield reported by `get_key_info()`.

In both cases, the bitfield is an array of 16 bytes,

```
uint8 states[16];
```

with one bit standing for each key on the keyboard. Bits are numbered from left to right, beginning with the first byte in the array, as illustrated below:



Bit numbers start with 0 and match key codes. For example, bit 0x3c corresponds to the **a** key, 0x3d to the **s** key, 0x3e to the **d** key, and so on. The first bit is 0x00, which doesn't correspond to any key. The first meaningful bit is 0x01, which corresponds to the **Escape** key.

When a key is down, the bit corresponding to its key code is set to 1. Otherwise, the bit is 0. However, for the three keys that toggle keyboard locks—**Caps Lock** (key 0x3b), **Num Lock** (key 0x22), and **Scroll Lock** (key 0x0f)—the bit is set to 1 if the lock is on and set to 0 if the lock is off, regardless of the state of the key itself.

To test the bitfield against a particular key,

- Select the byte in the `states` array that contains the bit for that key,
- Form a mask for the key that can be compared to that byte, and
- Compare the byte to the mask.

For example:

```
if (states[keyCode>>3] & (1 << (7 - (keyCode%8)))) {
   /* the key is down */
}
```

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Keyboard Information

This chapter covers everything you need to know about getting keyboard events, processing keyboard events, and learning about and changing the configuration of the user's keyboard.

The "Definitions" section provides definitions of some keyboard-related terminology, and describes some basic concepts, including the different kinds of keys, an overview of keyboard mapping, and modifier keys.

"Modifier Keys" discusses modifier keys in more detail.

"More on Keyboard Mapping" discusses keyboard mapping in more detail.

"Character Constants" covers the constants used to idenfity special characters and other special keys, such as the function keys.

"The Journey of a Keystroke" covers the process by which a keystroke makes its way from the keyboard to an application in BeOS.

The "Keyboard Messages" section lists the keyboard messages that get generated when the user presses or releases a key.

The "Key Codes" section provides information about the key codes used to represent each key on the keyboard, and discusses the functions that allow you to read the keyboard

state, as well as setting and changing the mapping between key codes and characters.

The "Key States" section discusses the ability to determine at any time which keys are down and which are not, using a bit-array of key states.

The "Input Functions" section (from the Input Server chapter) lists all keyboard-related functions, and includes specifics on how to read and understand the keymap-related structures.

The Be Book **Keyboard** **Keyboard Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

More on Keyboard Mapping

The key map records character values using the UTF-8 encoding of the Unicode Standard, making it possible to map keys to characters in any of the world's scripts. UTF-8 encodes 16-bit Unicode values in a variable number of bytes (from one to four). The main benefit to UTF-8 is that one-byte UTF characters $00 through $7F are identical to the ASCII standard that's been around for decades.

A `B_KEY_DOWN` message holds the character mapped to the key the user pressed as an array of bytes named, simply, "byte". The array is passed as a string to the `KeyDown()` hook function along with a count of the number of bytes in the string:

```
virtual void KeyDown(const char *bytes, int32 numBytes)
```

See "Character Encoding" in the "Responding to the User" section of the Interface Kit for a description of UTF-8 encoding and `get_key_map()` for an explanation of the key map.

Most keys are mapped to more than one character. The precise character that the key produces depends on which modifier keys are being held down and which lock states the keyboard is in at the time the key is pressed.

A few examples are given in the table below:

Key	No modifiers	Shift alone	Option alone	Shift + Option	Control
0x15	4	$			4
0x18	7	&	xa6	xa4	7
0x26	`B_TAB`	`B_TAB`	`B_TAB`	`B_TAB`	`B_TAB`
0x2e	i	I			`B_TAB`
0x40	g	G	"		0x07
0x43	k	K			`B_PAGE_UP`
0x51	n	N	ñ	Ñ	0x0e
0x55	/	?	xb8	xc0	/
0x64	`B_INSERT`	0	`B_INSERT`	0	`B_INSERT`

The mapping follows some fixed rules, including these:

- If a **Command** key is held down, the **Control** keys are ignored. **Command** trumps **Control**. Otherwise, **Command** doesn't affect the character that's reported for the key. If only **Command** is held down, the character that's reported is the same as if no modifiers were down; if **Command** and **Option** are held down, the character that's reported is the same as for **Option** alone; and so on.
- If a **Control** key is held down (without a **Command** key), **Shift**, **Option**, and all keyboard locks are ignored. Control trumps the other modifiers (except for **Command**).

- **Num Lock** applies only to keys on the numerical keypad. While this lock is on, the effect of the **Shift** key is inverted. **Num Lock** alone yields the same character that's produced when a **Shift** key is down (and **Num Lock** is off). **Num Lock** plus **Shift** yields the same character that's produced without either **Shift** or the lock.

- **Menu** and **Scroll Lock** play no role in determining how keys are mapped to characters.

The default key map also follows the conventional rules for **Caps Lock** and **Control**:

- **Caps Lock** applies only to the 26 alphabetic keys on the main keyboard. It serves to map the key to the same character as **Shift**. Using **Shift** while the lock is on undoes the effect of the lock; the character that's reported is the same as if neither **Shift** nor **Caps Lock** applied. For example, **Shift**+**g** and **Caps Lock**+**g** both are mapped to uppercase 'G', but **Shift**+**Caps Lock**+**g** is mapped to lowercase 'g'.

- However, if the lock doesn't affect the character, **Shift** plus the lock is the same as **Shift** alone. For example, **Caps Lock**+**7** produces '7' (the lock is ignored) and **Shift**+**7** produces '&' (Shift has an effect), so **Shift**+**Caps Lock**+**7** also produces '&' (only **Shift** has an effect).

- When **Control** is used with a key that otherwise produces an alphabetic character, the character that's reported has a value 0x40 less than the value of the uppercase version of the character (0x60 less than the lowercase version of the character). This often results in a character that is produced independently by another key. For example, **Control**+**i** produces the `B_TAB` character and **Control**+**l** produces `B_PAGE_DOWN`.

- When **Control** is used with a key that doesn't produce an alphabetic character, the character that's reported is the same as if no modifiers were on. For example, **Control**+**7** produces a '7'.

The Be Book Keyboard Keyboard Index

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRoster

Derived from: *none*
Declared in: be/app/Roster.h
Library: libbe.so
Summary: *more...*

The BRoster object represents a service that keeps a roster of all applications currently running. It can provide information about any of those applications, activate one of them, add another application to the roster by launching it, or get information about an application to help you decide whether to launch it.

There's just one roster and it's shared by all applications. When an application starts up, a BRoster object is constructed and assigned to a global variable, **be_roster**. You always access the roster through this variable; you never have to instantiate a BRoster in application code.

The BRoster identifies applications in three ways:

- By **entry_ref** references to the executable files where they reside.
- By their signatures. The signature is a unique identifier for the application assigned as a file-system attribute or resource at compile time or by the BApplication constructor at run time. You can obtain signatures for the applications you develop by contacting Be's developer support staff. They can also tell you what the signatures of other applications are.
- At run time, by their **team_id**s. A team is a group of threads sharing an address space; every application is a team.

If an application is launched more than once, the roster will include one entry for each instance of the application that's running. These instances will have the same signature, but different team identifiers.

Constructor and Destructor

BRoster()

```
BRoster(void)
```

Sets up the object's connection to the roster service.

When an application constructs its BApplication object, the system constructs a BRoster object and assigns it to the **be_roster** global variable. A BRoster is therefore readily available from the time the application is initialized until the time it quits; you don't have to construct one. The constructor is public only to give programs that don't have BApplication objects access to the roster.

~BRoster()

```
~BRoster()
```

Does nothing.

Member Functions

ActivateApp()

```
status_t ActivateApp(team_id team) const
```

Activates the *team* application (by bringing one of its windows to the front and making it the active window). This function works only if the target application has a window on-screen. The newly activated application is notified with a `B_APP_ACTIVATED` message.

See also: `BApplication::AppActivated()`

AddToRecentDocuments() , GetRecentDocuments()

```
void AddToRecentDocuments(const entry_ref *document,
    const char *appSig = NULL) const
```

```
void GetRecentDocuments(BMessage *refList, int32 maxCount,
    const char *ofType = NULL,
    const char *openedByAppSig = NULL) const
void GetRecentDocuments(BMessage *refList, int32 maxCount,
    const char *ofTypeList[] = NULL, int32 ofTypeListCount,
    const char *openedByAppSig = NULL) const
```

`AddToRecentDocuments()` adds the document file specified by *document* to the list of recent documents. If you wish to record that a specific application used the document, you can specify the signature of that application using the *appSig* argument; otherwise you can specify **NULL**.

`GetRecentDocuments()` returns a list of the most recent documents. The BMessage *refList* will be filled out with information about the *maxCount* most recently used documents. If you want to obtain a list of documents of a specific type, you can specify a pointer to that MIME type string in the *ofType* argument. Likewise, if you're only interested in files that want to be opened by a specific application, specify that application's signature in *openedByAppSig*; if you don't care, pass **NULL**.

If you want to get a list of files of multiple types, you can specify a pointer to an array of strings in *ofTypeList*, and the number of types in the list in *ofTypeListCount*.

Specifying **NULL** for *ofType* will fetch all files of all types.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting list of files.

AddToRecentFolders() , GetRecentFolders()

```
void AddToRecentFolders(const entry_ref *folder,
    const char *appSig = NULL) const
void GetRecentFolders(BMessage *refList, int32 maxCount,
    const char *openedByAppSig = NULL) const
```

`AddToRecentFolders()` adds the folder specified by *folder* to the list of recent folders. If you wish to record that a specific application used the folder, you can specify the signature of that application using the *appSig* argument; otherwise you can use **NULL**.

`GetRecentFolders()` returns a list of the most recently-accessed folders. The BMessage *refList* will be filled out with information about the *maxCount* most recently used folders. If you're only interested in folders that were used by a specific application, specify that application's signature in *openedByAppSig*; if you don't care, pass **NULL**.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting list of folders.

Broadcast()

```
status_t Broadcast(BMessage *message) const
status_t Broadcast(BMessage *message, BMessenger reply_to) const
```

Sends the *message* to every running application, except to those applications (`B_ARGV_ONLY`) that don't accept messages. The message is sent asynchronously with a timeout of 0. As is the case for other message-sending functions, the caller retains ownership of the *message*.

This function returns immediately after setting up the broadcast operation. It doesn't wait for the messages to be sent and doesn't report any errors encountered when they are. It returns an error only if it can't start the broadcast operation. If successful in getting the operation started, it returns `B_OK`.

Replies to the broadcasted message will be sent via the *reply_to* BMessenger, if specified. If *reply_to* is absent, the replies will be lost.

See also: `BMessenger::SendMessage()`

FindApp()

```
status_t FindApp(const char *type, entry_ref *app) const
status_t FindApp(entry_ref *file, entry_ref *app) const
```

Finds the application associated with the MIME data *type* or with the specified *file*, and modifies the *app* `entry_ref` structure so that it refers to the executable file for that application. If the *type* is an application signature, this function finds the application that has that signature. Otherwise, it finds the preferred application for the type. If the *file* is an application

executable, FindApp() merely copies the file reference to the *app* argument. Otherwise, it finds the preferred application for the file type.

In other words, this function goes about finding an application in the same way that Launch() finds the application it will launch.

If it can translate the *type* or *file* into a reference to an application executable, FindApp() returns B_OK. If not, it returns an error code, typically one describing a file system error.

See also: Launch()

GetAppInfo() , GetRunningAppInfo() , GetActiveAppInfo()

status_t **GetAppInfo(**const char **signature*, app_info **appInfo***) const status_t **GetAppInfo(**entry_ref **executable*, app_info **appInfo***) const
status_t **GetRunningAppInfo(**team_id *team*, app_info **appInfo***) const
status_t **GetActiveAppInfo(**app_info **appInfo***) const

These functions return (in *appInfo*) information about a specific application. In all cases, the application must be running.

- `GetAppInfo()` finds an app that has the given *signature*, or that was launched from the *executable* file. If there's more than one such app, the function chooses one at random.
- GetRunningAppInfo() reports on the app that corresponds to the given *team* identifier.
- GetActiveAppInfo() reports on the currently active app.

If they're able to fill in the app_info structure with meaningful values, these functions return B_OK. GetActiveAppInfo() returns B_ERROR if there's no active application. GetRunningAppInfo() returns B_BAD_TEAM_ID if *team* isn't a valid team identifier for a running application. GetAppInfo() returns B_ERROR if the application isn't running.

The app_info structure contains the following fields:

thread_id **thread**
The identifier for the application's main thread of execution, or –1 if the application isn't running. (The main thread is the thread in which the application is launched and in which its `main()` function runs.)

team_id **team**
The identifier for the application's team, or –1 if the application isn't running. (This will be the same as the *team* passed to GetRunningAppInfo().)

port_id port
The port where the application's main thread receives messages, or –1 if the application isn't running.

uint32 **flags**
A bitfield that contains information about the behavior of the application.

entry_ref **ref**
A reference to the file that was, or could be, executed to run the application. (This will be the same as the *executable* passed to GetAppInfo().)

char **signature** []
The signature of the application. (This will be the same as the *signature* passed to GetAppInfo().)

The **flags** bitfield can be tested (with the bitwise & operator) against these two constants:

- B_BACKGROUND_APP. The application won't appear in the Deskbar's application list.
- B_ARGV_ONLY. The application can't receive messages. Information can be passed to it at launch only, in an array of argument strings (as on the command line).

flags also contains a value that explains the application's launch behavior. This value is retrieved by masking **flags** with the B_LAUNCH_MASK constant. For example:

```
unit32 behavior = theInfo.flags & B_LAUNCH_MASK;
```

The result will match one of these three modes:

- B_EXCLUSIVE_LAUNCH. The application can be launched only if an application with the same signature isn't already running.
- B_SINGLE_LAUNCH. The application can be launched only once from the same executable file. However, an application with the same signature might be launched from a different executable. For example, if the user copies an executable file to another directory, a separate instance of the application can be launched from each copy.
- B_MULTIPLE_LAUNCH. There are no restrictions. The application can be launched any number of times from the same executable file.

These modes affect BRoster's Launch() function. Launch() can always start up a B_MULTIPLE_LAUNCH application. However, it can't launch a B_SINGLE_LAUNCH application if a running application was already launched from the same executable file. It can't launch a B_EXCLUSIVE_LAUNCH application if an application with the same signature is already running.

See also: Launch(), BApplication::GetAppInfo()

GetAppList()

```
void GetAppList(BList *teams) const
void GetAppList(const char *signature, BList *teams) const
```

Fills in the *teams* BList with team identifiers for applications in the roster. Each item in the list will be of type **team_id**. It must be cast to that type when retrieving it from the list, as follows:

```
BList *teams = new BList;
be_roster->GetAppList(teams);
team_id who = (team_id)teams->ItemAt(someIndex);
```

The list will contain one item for each instance of an application that's running. For example, if the same application has been launched three times, the list will include the **team_id**s for all three running instances of that application.

If a *signature* is passed, the list identifies only applications running under that signature. If a *signature* isn't specified, the list identifies all running applications.

See also: **TeamFor()**, the BMessenger constructor

GetRecentApps()

```
void GetRecentApps(BMessage *refList, int32 maxCount) const
```

GetRecentApps() returns a list of the most recently-launched applications. The BMessage *refList* will be filled out with information about the *maxCount* most recently-launched applications.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting applications.

GetRecentDocuments() see AddToRecentDocuments()

GetRecentFolders() see AddToRecentFolders()

IsRunning() see TeamFor()

Launch()

```
status_t Launch(const char *type,
    BMessage *message = NULL,
    team_id *team = NULL) const
status_t Launch(const char *type,
    BList *messages,
    team_id *team = NULL) const
status_t Launch(const char *type,
    int argc,
    char **argv,
    team_id *team = NULL) const
status_t Launch(const entry_ref *file,
    const BMessage *message = NULL,
    team_id *team = NULL) const
```

```
status_t Launch(const entry_ref *file,
    const BList *messages,
    team_id *team = NULL) const
status_t Launch(const entry_ref *file,
    int argc,
    const char * const char *argv,
    team_id *team = NULL) const
```

Launches the application associated with a MIME *type* or with a particular *file*. If the MIME *type* is an application signature, this function launches the application with that signature. Otherwise, it launches the preferred application for the type. If the *file* is an application executable, it launches that application. Otherwise, it launches the preferred application for the file type and passes the *file* reference to the application in a **B_REFS_RECEIVED** message. In other words, **Launch()** finds the application to launch just as **FindApp()** finds the application for a particular *type* or *file*.

If a *message* is specified, it will be sent to the application on-launch where it will be received and responded to before the application is notified that it's ready to run. Similarly, if a list of *messages* is specified, each one will be delivered on-launch. The caller retains ownership of the BMessage objects (and the container BList); they won't be deleted for you.

Sending an on-launch message is appropriate if it helps the launched application configure itself before it starts getting other messages. To launch an application and send it an ordinary message, call **Launch()** to get it running, then set up a BMessenger object for the application and call BMessenger's **SendMessage()** function.

If the target application is already running, **Launch()** won't launch it again, unless it permits multiple instances to run concurrently (it doesn't wait for the messages to be sent or report errors encountered when they are). It fails for **B_SINGLE_LAUNCH** and **B_EXCLUSIVE_LAUNCH** applications that have already been launched. Nevertheless, it assumes that you want the messages to get to the application and so delivers them to the currently running instance.

Instead of messages, you can launch an application with an array of argument strings that will be passed to its `main()` function. *argv* contains the array and *argc* counts the number of strings. If the application accepts messages, this information will also be packaged in a **B_ARGV_RECEIVED** message that the application will receive on-launch.

If successful, **Launch()** places the identifier for the newly launched application in the variable referred to by *team* and returns **B_OK**. If unsuccessful, it sets the *team* variable to –1 and returns an error code, typically one of the following:

- **B_BAD_VALUE**. The *type* or *file* is not valid, or an attempt is being made to send an on-launch message to an application that doesn't accept messages (that is, to a **B_ARGV_ONLY** application).
- **B_ALREADY_RUNNING.**, The application is already running and can't be launched again (it's a **B_SINGLE_LAUNCH** or **B_EXCLUSIVE_LAUNCH** application).

- **B_LAUNCH_FAILED.**, The attempt to launch the application failed for some other reason, such as insufficient memory.
- A file system error. The *file* or *type* can't be matched to an application.

See also: the BMessenger class, **GetAppInfo()**, **FindApp()**

StartWatching() , StopWatching()

status_t **StartWatching(**BMessenger *target*, uint32 *events* = B_REQUEST_LAUNCHED | B_REQUEST_QUIT**)** const
status_t **StopWatching(**BMessenger *target***)** const

StartWatching() initiates the application event monitor, which is used for keeping track of events such as application launches. The caller specifies the events to monitor through the *events* argument; *target* is the BMessenger to which the corresponding notification messages are sent. The *events* flags and the corresponding messages are listed below:

Flag	Message
B_REQUEST_LAUNCHED	B_SOME_APP_LAUNCHED
B_REQUEST_QUIT	B_SOME_APP_QUIT
B_REQUEST_ACTIVATED	B_SOME_APP_ACTIVATED

The fields in a notification message describe the application that was launched, quit, or activated:

Field	Type	Description
"mime_sig"	B_STRING_TYPE	MIME signature
"team"	B_INT32_TYPE	team_id
"thread"	B_INT32_TYPE	thread_id
"flags"	B_INT32_TYPE	application flags
"ref"	B_REF_TYPE	executable's entry_ref

StopWatching() terminates the application monitor previously initiated for a given BMessenger.

StartWatching()

TeamFor() , IsRunning()

team_id **TeamFor(**const char **signature***)** const team_id **TeamFor(**entry_ref **executable***)** const
bool **IsRunning(**const char **signature***)** const

bool **IsRunning(**entry_ref **executable***)** const

Both these functions query whether the application identified by its *signature* or by a reference to its *executable* file is running. **TeamFor()**`StartWatching()` returns its team identifier if it is, and `B_ERROR` if it's not. `IsRunning()` returns `true` if it is, and `false` if it's not.

If the application is running, you probably will want its team identifier (to set up a BMessenger, for example). Therefore, it's most economical to simply call `TeamFor()` and forego `IsRunning()`.

If more than one instance of the *signature* application is running, or if more than one instance was launched from the same *executable* file, `TeamFor()` arbitrarily picks one of the instances and returns its `team_id`.

See also: `GetAppList()`

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRoster Function Summary

Function Summary

Function	Description
ActivateApp()	Activates a particular application
AddToRecentDocuments()	Add a file to the recent documents list
AddToRecentFolders()	Add a folder to the recent folders list
Broadcast()	Sends a message to all running applications
FindApp()	Finds the application associated with a given MIME type or entry_ref
GetActiveAppInfo()	Returns information about the active application
GetAppInfo()	Returns information about a given application
GetAppList()	Returns a list of all running applications
GetRecentApps()	Get the list of recent applications
GetRecentDocuments()	Get the list of recent documents
GetRecentFolders()	Get the list of recent folders
GetRunningAppInfo()	Returns information about a given running application
IsRunning()	Tests whether a given application is running
Launch()	Launches an application

	associated with a given MIME type or entry_ref
BRoster()	Constructor
~BRoster()	Destructor
StartWatching()	Begin monitoring application events
StopWatching()	Stop monitoring application events
TeamFor()	Returns the team_id for a given application

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageRunner

Derived from: (none)
Declared in: be/app/MessageRunner.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

The BMessageRunner class provides a handy mechanism for automatically sending an arbitrary message to a BMessenger at specified intervals. The application that creates the BMessageRunner can specify the message, the BMessenger to send the message to, how often to send the message, and how many times it should be sent.

The system roster handles actually dispatching the messages to the appropriate BMessengers at the desired time intervals; this class simply acts as an intermediary through which your application asks the roster to schedule sending the messages.

Constructor and Destructor

BMessageRunner()

```
BMessageRunner(BMessenger target, const BMessage *message,
    bigtime_t interval, int32 count = -1)
BMessageRunner(BMessenger target, const BMessage *message,
    bigtime_t interval, int32 count, BMessenger replyTo)
```

Tells the roster to send the specified *message* to the *target* BMessenger every *interval* microseconds. The message will be sent *count* times (if *count* is -1, the message will be sent forever, or until the BMessageRunner is reconfigured or deleted).

The second form of the constructor lets the application specify, in *replyTo*, the BMessenger to which replies to the message should be sent.

The BMessageRunner can be reconfigured (to change the *interval* or *count*) by calling `SetInterval()` and `SetCount()`.

After constructing a BMessageRunner, you should call `InitCheck()` to ensure that the object was created properly.

~BMessageRunner()

```
virtual ~BMessageRunner()
```

Asks the roster to stop sending the message.

Member Functions

GetInfo()

```
status_t GetInfo(bigtime_t *interval, int32 *count) const
```

`GetInfo()` returns in *interval* the time in microseconds that will pass between messages being sent, and in *count* the number of times the message will be sent.

RETURN CODES

- B_OK. Information returned successfully.
- **`B_NAME_NOT_FOUND`**. The roster returned invalid information about the BMessenger.
- **`B_BAD_VALUE`**. The roster returned invalid information about the BMessenger.
- Other errors. In general, getting an error back from this function is a bad thing.

InitCheck()

```
status_t InitCheck(void) const
```

`InitCheck()` returns a result code indicating **`B_OK`** if the BMessageRunner constructor executed sucessfully, or some other value if an error occurred setting up the object. You should call this immediately after creating a BMessageRunner, and shouldn't use the object if this function returns anything but **`B_OK`**.

SetCount() , SetInterval()

```
status_t SetCount(int32 count)
status_t SetInterval(bigtime_t interval)
```

`SetCount()` sets the number of times the BMessageRunner will send the message. If you want the message to be sent forever (until the object is deleted or `SetCount()` is called again), specify -1.

`SetInterval()` sets the number of microseconds that will pass between messages being sent.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageRunner Function Summary

Function Summary

Function	Description
GetInfo()	Returns the current interval and count values
InitCheck()	Returns information about the success of the constructor
BMessageRunner()	Constructor
~BMessageRunner()	Destructor
SetCount()	Sets the number of times the message will be sent
SetInterval()	Sets the delay between message transmissions

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BCursor

Derived from: none
Mix-in classes: BArchivable
Declared in: be/app/Cursor.h
Library: libbe.so
Summary: *more...*

You can use a BCursor to represent a mouse cursor as an object instead of as a block of pixel data; this can be more convenient in some situations. Also, if you want to call `BApplication::SetCursor()` without forcing an immediate sync of the Application Server, you have to use a BCursor.

The default BCursors are **`B_CURSOR_SYSTEM_DEFAULT`** for the hand cursor and **`B_CURSOR_I_BEAM`** for the I-beam text editing cursor.

Cursor Data Format

The first four bytes of cursor data give information about the cursor:

- Byte 1: *Size in pixels-per-side*. Cursors are always square; currently, only 16-by-16 cursors are allowed.
- Byte 2: *Color depth in bits-per-pixel*. Currently, only one-bit monochrome is allowed.
- Bytes 3&4: *Hot spot coordinates*. Given in "cursor coordinates" where (0,0) is the upper left corner of the cursor grid, byte 3 is the hot spot's y coordinate, and byte 4 is its x coordinate. The hot spot is a single pixel that's "activated" when the user clicks the mouse. To push a button, for example, the hot spot must be within the button's bounds.

Next comes an array of pixel color data. Pixels are specified from left to right in rows starting at the top of the image and working downward. The size of an array element is the depth of the image. In one-bit depth...

- the 16x16 array fits in 32 bytes.
- 1 is black and 0 is white.

Then comes the pixel transparency bitmask, given left-to-right and top-to-bottom. 1 is opaque, 0 is transparent. Transparency only applies to white pixels—black pixels are always opaque.

Constructor and Destructor

BCursor()

```
BCursor(const void *cursorData)
BCursor(BMessage *archive)
```

Initializes the new cursor object. If you specify a non-**NULL** value for *cursorData*, the cursor is initialized with the specified cursor data.

If you specify a **NULL** value for *cursorData*, the cursor is useless; since this class doesn't currently provide a means of setting the cursor data once the object is instantiated, you're out of luck, so why bother?

BCursor doesn't currently implement archiving, so you shouldn't use the second form.

~BCursor()

```
virtual ~BCursor()
```

Releases any resources used by the cursor.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive**)

Not currently implemented; always returns **NULL**.

See also: **BArchivable::Instantiate()**, **instantiate_object()**, **Archive()**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BCursor Function Summary

Function Summary

Function	Description
BCursor()	Constructor
~BCursor()	Destructor
Instantiate()	Not implemented: Creates a BCursor from a BMessage

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BControl Function Summary

Function Summary

Function	Description
Archive()	Archives a BControl into a BMessage
AttachedToWindow()	Provided for implementation
BControl()	Constructor
~BControl()	Destructor
GetSupportedSuites()	Provided for implementation
Instantiate()	Creates a BControl from a BMessage
Invoke()	Dispatches a message to the designated target
IsEnabled()	Returns true if control enabled
IsFocusChanging()	protected: Returns true if redraw due to focus change.
KeyDown()	Provided for implementation
Label()	Returns text label
MakeFocus()	Provided for implementation
MessageReceived()	Provided for implementation
ResolveSpecifier()	Provided for implementation
SetEnabled()	Hook called to enable or disable control
SetLabel()	Sets text label
SetValue()	Hook called to set current control value
Value()	Returns current control value
WindowActivated()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRadioButton

Derived from: public BControl
Declared in: be/interface/RadioButton.h
Library: libbe.so
Summary: *more...*

A BRadioButton object draws a labeled, two-state button that's displayed in a group along with other similar buttons. The button itself is a round icon that has a filled center when the BRadioButton is turned on, and is empty when it's off. The label appears next to the icon.

Only one radio button in the group can be on at a time; when the user clicks a button to turn it on, the button that's currently on is turned off. One button in the group must be on at all times; the user can turn a button off only by turning another one on. The button that's on has a value of 1 (`B_CONTROL_ON`); the others have a value of 0 (`B_CONTROL_OFF`).

The BRadioButton class handles the interaction between radio buttons in the following way: A direct user action can only turn on a radio button, not turn it off. However, when the user turns a button on, the BRadioButton object turns off all sibling BRadioButtons—that is, all BRadioButtons that have the same parent as the one that was turned on.

This means that a parent view should have no more than one group of radio buttons among its children. Each set of radio buttons should be assigned a separate parent—perhaps an empty BView that simply contains the radio buttons and does no drawing of its own.

Constructor and Destructor

BRadioButton()

```
BRadioButton(BRect frame, const char *name,
    const char *label,
    BMessage *message,
    uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    uint32 flags = B_WILL_DRAW | B_NAVIGABLE)
BRadioButton(BMessage *archive)
```

Initializes the BRadioButton by passing all arguments to the BControl constructor without change. BControl initializes the radio button's *label* and assigns it a model *message* that identifies the action that should be taken when the radio button is turned on. When the user turns the button on, the BRadioButton posts a copy of the *message* so that it can be delivered to the target handler.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed without change from BControl to the BView constructor.

The BRadioButton draws at the bottom of its frame rectangle beginning at the left side. It ignores any extra space at the top or on the right. (However, the user can click anywhere within the *frame* rectangle to turn on the radio button). When the object is attached to a window, the height of the rectangle will be adjusted so that there is exactly the right amount of room to accommodate the label.

See also: the BControl and BView constructors, `AttachedToWindow()`

~BRadioButton()

```
virtual ~BRadioButton()
```

Does nothing; a BRadioButton doesn't need to clean up after itself when it's deleted.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage ****archive*)

Returns a new BRadioButton object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the message doesn't contain data for an archived BRadioButton object, this function returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Member Functions

Archive()

virtual status_t **Archive(**BMessage **archive*, bool *deep* = true**)** const

Calls the inherited version of **Archive()** and doesn't add anything specific to the BRadioButton class to the BMessage *archive*.

See also: **BArchivable::Archive()**, **Instantiate()** static function

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of **AttachedToWindow()** to set the view and low colors of the BRadioButton to the match its parent's view color, and to resize the radio button vertically to fit the height of the label it displays. The height of the label depends on the BRadioButton's font.

See also: **BControl::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the radio button—the circular icon—and its label. The center of the icon is filled when the BRadioButton's value is 1 (**B_CONTROL_ON**); it's left empty when the value is 0 (**B_CONTROL_OFF**).

See also: **BView::Draw()**

GetPreferredSize()

virtual void **GetPreferredSize(**float **width*, float **height***)**

Calculates the optimal size for the radio button to display the icon and the label in the current font, and places the result in the variables that the *width* and *height* arguments refer to. **ResizeToPreferred()**, defined in the BView class, resizes a view's frame rectangle to the preferred size, keeping its left and top sides constant. **AttachedToWindow()** automatically resizes a radio button to its preferred height, but doesn't modify its width.

See also: **BView::GetPreferredSize()**, **AttachedToWindow()**

KeyDown()

virtual void **KeyDown(**const char ***bytes*, int32 *numBytes***)**

Augments the inherited versions of **KeyDown()** to turn the radio button on and deliver a message to the target BHandler when the character passed in *bytes* is **B_SPACE** or **B_ENTER**.

See also: **BView::KeyDown()**, **SetValue()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event in the radio button by tracking the cursor while the user holds the mouse button down. If the cursor is pointing to the radio button when the user releases the mouse button, this function turns the button on (and consequently turns all sibling BRadioButtons off), calls the BRadioButton's **Draw()** function, and posts a message that will be delivered to the target BHandler. Unlike a BCheckBox, a BRadioButton posts the message—it's "invoked"—only when it's turned on, not when it's turned off.

See also: **BControl::Invoke()**, **BControl::SetTarget()**, **SetValue()**

SetValue()

virtual void **SetValue(**int32 *value***)**

Augments the BControl version of **SetValue()** to turn all sibling BRadioButtons off (set their values to 0) when this BRadioButton is turned on (when the *value* passed is anything but 0).

See also: **BControl::SetValue()**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRadioButton Function Summary

Function Summary

Function	Description
Archive()	Archives a BRadioButton into a BMessage
AttachedToWindow()	Provided for implementation
Draw()	Provided for implementation
GetPreferredSize()	Provided for implementation
Instantiate()	Creates a BRadioButton from a BMessage
KeyDown()	Provided for implementation
MouseDown()	Provided for implementation
BRadioButton()	Constructor
~BRadioButton()	Destructor
SetValue()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSlider Function Summary

Function Summary

Function	Description
AttachedToWindow()	Provided for implementation
BarColor()	Returns the color of the slider bar
BarFrame()	Returns the frame rectangle enclosing the slider bar
BarThickness()	Returns the slider bar's thickness
Draw()	Provided for implementation
DrawBar()	Draws the slider bar
DrawFocusMark()	Draws the slider focus mark if it is in focus
DrawHashMarks()	Draws the hash marks
DrawSlider()	Draws the slider
DrawText()	Draws the slider's text areas
DrawThumb()	Draws the slider's thumb
FillColor()	Returns the fill color
FrameResized()	Provided for implementation
GetPreferredSize()	Provided for implementation
HashMarkCount()	Returns the number of hash marks displayed
HashMarks()	Returns the placement of the hash marks
HashMarksFrame()	Returns the frame rectangle enclosing the hash marks

KeyDown()	Provided for implementation
KeyIncrementValue()	Returns the increment value from to keyboard control
MaxLimitLabel()	Returns the label marking the maximum slider value
MinLimitLabel()	Returns the label marking the minimum slider value
ModificationMessage()	Returns the slider modification message
MouseDown()	Provided for implementation
OffscreenView()	Returns a pointer to the offscreen BView used in drawing the BSlider
Orientation()	Returns the slider's orientation
Position()	Returns the value of the slider
ResizeToPreferred()	Provided for implementation
SetBarColor()	Sets the color of the slider bar
SetBarThickness()	Sets the slider bar's thickness
SetFont()	Sets the font used for the slider's labels
SetHashMarkCount()	Sets the number of hash marks displayed
SetHashMarks()	Sets the placement of the hash marks
SetKeyIncrementValue()	Sets the increment value from to keyboard control
SetLimitLabels()	Sets the labels marking the extreme values on the slider

SetModificationMessage()	Sets the slider modification message
SetOrientation()	Sets the slider's orientation
SetPosition()	Sets the value of the slider
SetSnoozeAmount()	Sets the snooze period between updates during tracking
SetStyle()	Sets the thumb style
SetValue()	Sets the value of the slider
BSlider()	Constructor
~BSlider()	Destructor
SnoozeAmount()	Returns the snooze period between updates during tracking
Style()	Returns the thumb style
ThumbFrame()	Returns the frame rectangle enclosing the thumb
UpdateText()	Hook returning status text to display with the slider
UseFillColor()	Sets the fill color
Value()	Returns the value of the slider
ValueForPoint()	Returns the slider value given screen coordinates

---

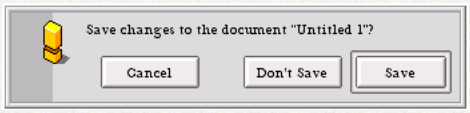
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAlert

Derived from: BWindow > BLooper > BHandler
Mix-in classes: BArchivable
Declared in: be/interface/Alert.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

A BAlert displays a modal window that notifies the user of an error (or the like), and provides a set of buttons (three buttons, max) that lets the user respond to the situation. For example, here's a typical "unsaved changes" alert:



When the user clicks one of the buttons, the alert panel is automatically removed from the screen, the index of the chosen button (0,1, or 2, left to right) is reported to your app, and the BAlert object is deleted.

The buttons are automatically aligned within the panel (as shown above). The rightmost button is the *default button*—i.e., it's mapped to the **Enter** key. You can assign your own shortcuts through the `SetShortcut()` function (*don't* use `BWindow::AddShortcut()`).

Construction and Deletion

BAlert objects must be constructed with **new**; you can't allocate a BAlert on the stack.

A BAlert object deletes itself when it's removed from the screen. You never need to delete the BAlert objects that you display.

Creating and Running an Alert Panel

The following code creates and displays the alert panel shown above:

```
BAlert *myAlert = new BAlert("title", "Save changes to ...")
   "Cancel", "Don't save", "Save",
   B_WIDTH_AS_USUAL, B_OFFSET_SPACING, B_WARNING_ALERT);
myAlert->SetShortcut(0, B_ESCAPE);
   int32 button_index = alert->Go();
```

This is the canonical "Do it/Don't do it/Cancel" alert. Any alert that has a "Cancel" button should map the Escape key as a shortcut, as shown here.

The `Go()` function runs the panel: It displays the panel, removes the panel when the user is done, and returns the index of the button that the user clicked.

Asynchronous Alerts

The default (no argument) version of `Go()` shown above is synchronous: It doesn't return until the user clicks a button. There's also an asynchronous version of `Go()` that returns immediately and (optionally) sends back the user's response in a BMessage. See `Go()` for details.

Look and Feel

By default, a BAlert object uses the `B_MODAL_APP_WINDOW_FEEL`. This means that it blocks your application's other windows. If you want to broaden the feel so it blocks all windows (`B_MODAL_ALL_WINDOW_FEEL`), or restrict it so it blocks only a few of your app's windows (`B_MODAL_SUBSET_WINDOW_FEEL`), call `BWindow::SetFeel()`. In the subset case, you'll also have to call `BWindow::AddToSubset()`.

Never change the object's look (`B_MODAL_WINDOW_LOOK`).

Constructor

BAlert()

```
BAlert(const char *title,
    const char *text,
    const char *button0Label,
    const char *button1Label = NULL,
    const char *button2Label = NULL,
    button_width widthStyle = B_WIDTH_AS_USUAL,
    alert_type type = B_INFO_ALERT)
BAlert(const char *title,
    const char *text,
    const char *button0Label,
    const char *button1Label,
    const char *button2Label,
    button_width widthStyle,
    button_spacing spacing,
    alert_type type = B_INFO_ALERT)
BAlert(BMessage *archive)
```

Creates an alert panel that's ready to be displayed. The arguments are:

- *title* is the name of the panel. Since an alert panel doesn't have a title tab, the title isn't actually displayed in the panel. Nonetheless, it can be useful in debugging: The title is used to name the window thread ("*w*>title").
- *text* is displayed in the body of the panel; it should describe the reason the panel is being shown ("Are you sure you want to quit?", "Invalid name.", etc.).
- The *buttonLabel* arguments label the panel's buttons. An alert panel can have one, two, or three buttons—however many labels you supply is the number of buttons that are displayed. The buttons are arranged at the bottom of the panel as shown in the illustration above, and each is assigned an index (0, 1, or 2, left to right) that's used to identify the button that the user clicked.
- *widthStyle* is a constant that describes how the buttons should be sized, one of `B_WIDTH_AS_USUAL`, `B_WIDTH_FROM_WIDEST`, or `B_WIDTH_FROM_LABEL`. See `button_width` for details.
- *spacing* is a constant that describes how the buttons are spaced, either **`B_EVEN_SPACING`**, or **`B_OFFSET_SPACING`**. The constructor that lacks the *spacing* argument uses **`B_EVEN_SPACING`**. See `button_spacing` for details.
- *type* is a constant that determines which of the five alert icons is displayed in the panel: **`B_EMPTY_ALERT`**, **`B_INFO_ALERT`**, **`B_IDEA_ALERT`**, **`B_WARNING_ALERT`**, or **`B_STOP_ALERT`**. See `alert_type` for pictures.

After the BAlert is constructed, Go() must be called to display it. The panel is removed and the object is deleted after the user clicks a button.

Member Functions

AlertPosition()

BPoint **AlertPosition(**float *width*, float *height***) const**

Computes the "best" frame for an alert of the given *width* and *height*, and returns the top left corner of the computed frame in screen coordinates. This function is called automatically when you construct your BAlert; you never have to invoke it yourself.

ButtonAt()

BButton ***ButtonAt(**int32 *index***)** const

Returns a pointer to the BButton object identified by *index*. Indices begin at 0 and count buttons from left to right. The BButton belongs to the BAlert object and should not be freed.

FrameResized() see BWindow::FrameResized()

Go()

int32 **Go(**void**)**
status_t Go(BInvoker ***invoker***)**

Displays the alert panel. `Go()` can operate synchronously or asynchronously:

- The no-argument version is synchronous: The function doesn't return until the user has clicked a button and the panel has been removed from the screen.. The value it returns is the index of the clicked button (0,1, or 2, left-to-right)..
- The BInvoker argument version is asynchronous: The function returns immediately (with **`B_OK`**) and the button index is delivered as the **`int32`** "which" field of the BMessage that's sent to the BInvoker's target.

If you call `Go()` with a (literal) **`NULL`** argument...

```
alert->Go(NULL);
```

...the asynchronous version is used, but the BMessage isn't sent.

The synchronous version deletes the object before it returns; the asynchronous version deletes it after the message is sent. In either case, you should consider the BAlert object to be invalid after you call `Go()`.

If the BAlert is sent a **B_QUIT_REQUESTED** message while the panel is still on-screen, the synchronous version of **Go()** returns -1; the asynchronous version suppresses the index-reporting message.

MessageReceived() see BWindow::MessageReceived()

SetShortcut() , Shortcut()

void **SetShortcut(**int32 *index*, char *shortcut***)**
char **Shortcut(**int32 *index***)** const

These functions set and return the *shortcut* character that's mapped to the button at *index*. A given button can have only one shortcut except for the rightmost button, which, in addition to the shortcut that you give it here, is always mapped to **B_ENTER**.

If you create a "Cancel" button, you should give it a shortcut of **B_ESCAPE**.

TextView()

BTextView ***TextView(**void**)** const

Returns a pointer to the BTextView object that contains the textual information that's displayed in the panel. You can fiddle with this object but you mustn't delete it.

Constants and Defined Types

alert_type

```
enum alert_type { B_EMPTY_ALERT,
    B_INFO_ALERT,
    B_IDEA_ALERT,
    B_WARNING_ALERT,
    B_STOP_ALERT };
```

The `alert_type` constants represent the five alert icons:

B_EMPTY_ALERT	B_INFO_ALERT	B_IDEA_ALERT	B_WARNING_ALERT	B_STOP_ALERT
(none)				

button_spacing

```
enum button_spacing { B_EVEN_SPACING,
    B_OFFSET_SPACING };
```

Used to set the spacing of the alert's buttons:

Constant	Meaning
B_EVEN_SPACING	The buttons are evenly spaced. (This is used by the constructor without the *spacing* argument.)
B_OFFSET_SPACING	If the alert has more than one button, the leftmost button is shifted left to separate it from its neighbor(s). You should use this configuration for an alert that has a (leftmost) "Cancel" button, as shown in the illustration at the beginning of this document.

Archived Fields

Field	Type code	Meaning
"_text"	B_STRING_TYPE	The alert's descriptive text.
"_atype"	B_INT32_TYPE	The alert_type value.
"_but_width"	B_INT32_TYPE	The button_width value.
"_but_key" (array)	B_INT32_TYPE	Keyboard shortcut values (one field per button).

In a deep copy, the following views appear in the "_views" field:

View name	Level	Meaning
"_master_"	0	Background view.
"_tv_"	1	BTextView for the alert panel's text.
"_b0_"	1	BButton with index 0.
"_b1_"	1	BButton with index 1.
"_b2_"	1	BButton with index 2.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BAlert Function Summary

Function Summary

Function	Description
BAlert()	Constructor
AlertPosition()	Returns the best frame for a given sized alert
ButtonAt()	Returns pointer to a given BButton in the panel
Go()	Runs the alert panel
SetShortcut()	Set shortcut character for a button
Shortcut()	Returns shortcut character for a button
TextView()	Returns a pointer to the alert's BTextView object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Scripting and Archival

Derived from: BHandler
Declared in: be/interface/View.h
Library: libbe.so
Summary: *more...*

This section describes BView's scripting and archival API.

Scripting Suites

The BView class implements the two scripting suites "suite/vnd.Be-view" and "suite/vnd.Be-container-view." "suite/vnd.Be-container-view" is only implemented if the view contains a shelf; the suite consists of the single property "Shelf."

Scripting Properties

- "Frame"

View's frame rectangle

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns the view's frame rectangle.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Sets the view's frame rectangle.

The "Frame" property represents the frame rectangle of the view. The frame is passed as a BRect (`B_RECT_TYPE`).

- "Hidden"

Visibility of the view

Message	Specifiers	Meaning
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns `true` if the view is hidden; `false` otherwise.
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Hides or shows the view.

The "Hidden" property determines the visibility of the view. The messages are equivalent to manipulating the view with the `IsHidden()`, `Hide()`, and `Show()`. Note that this differs slightly from BWindow's "Hidden" property, where multiple hides or shows are not nested.

The "Hidden" property determines the visibility of the view. The messages are equivalent to manipulating the view with the `IsHidden()`, `Hide()`, and `Show()` methods with one caveat:

nested **Hide()** and **Show()** calls are disabled so that multiple scripting Hide commands may be undone with a single Show.

"Shelf"

BShelf attached to the view

Message	Specifiers	Meaning
any	**B_DIRECT_SPECIFIER**	Directs the scripting message to the shelf.

The "Shelf" property pops the current specifier off the specifier stack and then passes the scripting message to the shelf. If no shelf is present, an error is returned.

"View"

Child views

Message	Specifiers	Meaning
B_COUNT_PROPERTIES	**B_DIRECT_SPECIFIER**	Returns the number of of child views.
any	**B_INDEX_SPECIFIER**, B_REVERSE_INDEX_SPECIFIER, B_NAME_SPECIFIER	Directs the scripting message to the

		specified view.

The "View" property represents the child views of the current view. For all messages except `B_COUNT_PROPERTIES`, the current specifier is popped off the specifier stack before the scripting message is passed to the target view. Views can be specified either by index (as found by **`ChildAt()`**), or name (as found by **`FindView()`**).

Archived Fields

The **Archive()** function adds the following fields to its BMessage argument:

Field	Type code	Meaning
"_frame"	**B_RECT_TYPE**	The view's frame rectangle.
"_resize_mode"	**B_INT32_TYPE**	Resizing mode.
"_flags"	**B_INT32_TYPE**	View flags.
"_fname" (array)	**B_STRING_TYPE**	Font family and style.
"_fflt" (array)	**B_FLOAT_TYPE**	Font size, shear, and rotation (-1 if default).
"_color" (array)	**B_INT32_TYPE**	High, low, and view colors.
"_evmask" (array)	**B_INT32_TYPE**	Event mask and options.
"_dbuf"	**B_INT32_TYPE**	Double buffering flag.
"_origin"	**B_POINT_TYPE**	Origin.
"_psize"	**B_FLOAT_TYPE**	Pen size.
"_ploc"	**B_POINT_TYPE**	Pen location.
"_lmcapjoin" (array)	**B_INT16_TYPE**	Line cap mode and join mode.
"_lmmiter"	**B_FLOAT_TYPE**	Line mode miter value.
"_blend" (array)	**B_INT16_TYPE**	Blending alpha and mode.
"_dmod"	**B_INT32_TYPE**	Drawing mode.
"_views"	**B_MESSAGE_TYPE**	Archived child views (deep

		copy only).

Some of these fields may not be present if the setting they represent isn't used, or is the default value. For example, if the font's family and style were not changed, the "_fname" field won't be found in the archive.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Listening to the User

When the user fiddles with the mouse or keyboard, the Application Server figures out which window the event is intended for, and then forms an *interface message* that it sends to the BWindow object. The BWindow receives the message in its **`DispatchMessage()`** function and invokes an *interface hook function* declared by BWindow or BView. If your application wants to respond to an event it simply implements the appropriate hook function. For example, if you want to watch for mouse down events within a view, you implement **`MouseDown()`** in your BView subclass.

You can also find out what's going on with the keyboard and mouse through the (global) **`get_key_info()`** function and the **`BView::GetMouse()`** function. You typically use these functions within the implementation of an interface hook function to get more information about the keyboard or mouse.

You *shouldn't* call **`get_key_info()`** or **`BView::GetMouse()`** in a loop to track the keyboard or mouse unless you really know what you're doing (some game developers may need to poll). To track the keyboard, you implement BView's **`KeyDown()`** and **`KeyUp()`** functions. To track the mouse, implement **`MouseDown()`**, **`MouseMoved()`**, and **`MouseUp()`**.

Interface Messages

The interface messages are reasonably self-explanatory. See the notes below and the documentation of the individual hook functions for more information.

Message	Hook function
B_ZOOM	BWindow::Zoom()
B_MINIMIZE	BWindow::Minimize()
B_KEY_DOWN	BView::KeyDown()
B_KEY_UP	BView::KeyUp()
B_MOUSE_DOWN	BView::MouseDown()
B_MOUSE_UP	BView::MouseUp()
B_MOUSE_MOVED	BView::MouseMoved()
B_VIEW_MOVED	BView::FrameMoved()
B_VIEW_RESIZED	BView::FrameResized()
B_VALUE_CHANGED	BScrollBar::ValueChanged()
B_WINDOW_ACTIVATED	BWindow::WindowActivated() BView::WindowActivated()
B_QUIT_REQUESTED	BLooper::QuitRequested()
B_WINDOW_MOVED	BWindow::FrameMoved()
B_WINDOW_RESIZED	BWindow::FrameResized()
B_SCREEN_CHANGED	BWindow::ScreenChanged()
B_WORKSPACE_ACTIVATED	BWindow::WorkspaceActivated()
B_WORKSPACES_CHANGED	BWindow::WorkspacesChanged()
B_PULSE	BView::Pulse()

Notes:

- B_ZOOM is usually caused by the user operating the zoom button in the window's title tab.
- B_MINIMIZE tells the window to hide/show itself. This message is usually caused by the user double-clicking the window's tab (to hide), and clicking the window token in the application's window list (in the DeskBar).
- B_KEY_DOWN and B_KEY_UP report that the user pressed and then release a *character key*. When the key is held down, repeated B_KEY_DOWN messages are sent. If a key is mapped to a string of characters, a **B_KEY_DOWN/B_KEY_UP** pair is generated for each character. Pressing a *modifier keys* on its own doesn't generate key messages.
- B_MOUSE_DOWN, B_MOUSE_MOVED, and B_MOUSE_UP are delivered (as hook functions) to the BView that the mouse is over at the time, regardless of where it started. For example, if the user drags the mouse through your view, you'll get a series of B_MOUSE_MOVED messages without getting a B_MOUSE_UP or a B_MOUSE_DOWN. The **BView::MouseMoved()** function has a code that

tells you whether the mouse is just entering, inside, or exiting your view. You can tell your BView to track the mouse even when it's outside your view through **BView::SetMouseEventMask()**.

- A **B_VALUE_CHANGED** message reports that the Application Server changed a value associated with a BScrollBar objects.
- **B_WORKSPACE_ACTIVATED** reports that the active workspace (the one displayed on-screen) has changed. The message is only sent to windows that live in one of the affected workspaces.
- **B_WORKSPACES_CHANGED** notifies the window that the set of workspaces in which it can be displayed has changed.
- A BWindow reinterprets a **B_QUIT_REQUESTED** message, originally defined for the BLooper class in the Application Kit, to mean a user request to close the window. However, it doesn't redeclare the **QuitRequested()** hook function that it inherits from BLooper.

Tracking the Keyboard

The view that's responsible for handling keyboard events is called as the *focus view*. The focus view is the view within the active window that's displaying the current selection, or the control that's marked to show that it can be operated from the keyboard. Only one view in the window can be in focus at a time.

You promote a view to focus by calling **BView::MakeFocus()**, and you ask for the current focus through **BWindow::CurrentFocus()**.

Focus View Commands

The table below maps a keyboard action to the hook function it invokes or the message it sends. In each of these cases, the function or message is sent to the *current* focus view (which may change between a key down and a key up)

User action	Function or message
Press a character key	**BView::KeyDown()**
Release a character key	**BView::KeyUp()**
Command+**a**	**B_SELECT_ALL**
Command+**c**	**B_SELECT_COPY**
Command+**x**	**B_SELECT_CUT**
Command+**v**	**B_SELECT_PASTE**

The focus view needn't respond to all of these functions and commands, but a BView that doesn't respond to any of them should never promote itself to focus.

Other Keyboard Commands

There are four other built-in keyboard commands:

User action	Message or Action	Target
Command+**w**	`B_QUIT_REQUESTED`	The active window.
Command+**q**	`B_QUIT_REQUESTED`	The active app.
Option+**Tab**	Changes focus to the next navigable view.	The active window.
Enter	Activates the default button.	The active window.

Kinds of Keyboard Messages

`B_KEY_UP` events are always assigned to the view that's in focus when the user releases the key—even if the previous `B_KEY_DOWN` message performed a shortcut, forced keyboard navigation, or was assigned to the default button.

Message Protocols

The event messages that the Application Server sends to a BWindow usually have more information in them than is passed to the corresponding hook function. For example, while a `B_MOUSE_DOWN` message knows where, when, and which mouse button was pressed (among other things), only the "where" information is passed to the `MouseDown()` function.

You can retrieve the message that initiated a hook function from within the hook function itself by calling BWindow::CurrentMessage() function:

```
void MyView::MouseDown(BPoint where)
{
   BMessage *msg = Window()->CurrentMessage();
   ...
```

The User Interface

The Interface Kit provides interface mechanisms that your classes can participate in, if they coordinate with kit-defined code. Two such mechanisms are described below—keyboard navigation and the drag-and-drop delivery of messages.

Keyboard Navigation

Keyboard navigation is a mechanism for allowing users to manipulate views—especially buttons, check boxes, and other control devices—from the keyboard. It gives users the ability to:

- Move the focus of keyboard actions from view to view within a window by pressing the **Tab** key.
- Operate the view that's currently in focus by pressing **spacebar** and **Enter** (to invoke it) or the arrow keys (to move around inside it).

The first ability—navigation between views—is implemented by the Interface Kit. The second—navigation within a view—is up to individual applications (although the BControl class helps a little), as are most view-specific aspects of the user interface. The only trick, and it's not a difficult one, is to make the two kinds of navigation work together.

To participate in the navigation mechanism, a class derived from BView needs to coordinate three aspects of its code—setting navigation flags, drawing an indication that the BView is in focus, and responding to keyboard events. The following sections discuss each of these elements.

Setting Navigation Flags

The **B_NAVIGABLE** flag marks a BView as an eligible target for keyboard navigation. It's one flag in a mask that the BView constructor sets, along with other view attributes. For example:

```
MyView::MyView(BRect frame, const char *name,
                        uint32 resizingMode, uint32 flags)
   : BView(frame, name, resizingMode, flags|B_NAVIGIBLE|B_WILL_DRAW)
{
   . . .
}
```

When the user presses the **Tab** key, the focus moves from one **B_NAVIGIBLE** target to the next, working first down and then across the view hierarchy. That is, if a BView has both **B_NAVIGIBLE** children and **B_NAVIGIBLE** siblings, the children will be targeted before the siblings.

The flag should be removed from the mask when the view is disabled or cannot become the focus view for any reason, and included again when it's re-enabled. The mask can be altered with the **SetFlags()** function:

```
if ( /* cannot become the focus view */ )
   SetFlags(Flags() & ~B_NAVIGIBLE);
else
   SetFlags(Flags() | B_NAVIGIBLE);
```

Most navigible BViews are control devices and derive from the BControl class. All BControls are navigible by default and BControl has a **SetEnabled()** function that turns the **B_NAVIGIBLE** flag on and off, so this work is already done for objects that inherit from BControl.

You may also want to set a view's **B_NAVIGIBLE_JUMP** flag to permit larger jumps between navigible views. **Control**+**Tab** moves the focus from one group of views to another, where the groups are (hopefully) obvious to the user from their arrangement in the window.

B_NAVIGIBLE_JUMP marks positions in the view hierarchy for these larger jumps. When the user presses **Control**+**Tab**, focus jumps to the next **B_NAVIGIBLE_JUMP** view. If a **B_NAVIGABLE_JUMP** view isn't also **B_NAVIGABLE**, focus moves to the next available **B_NAVIGABLE** view. For example, if a **B_NAVIGABLE_JUMP** parent view is not navigible itself but has navigible children, **Control**+**Tab** focusses on the first **B_NAVIGABLE** child.

Drawing the Focus Indicator

When the user navigates to a view, the BView needs to draw some sort of visual indication that it's the current focus for keyboard actions. Be-defined views underline text (for example, a button label) when the view is in focus, or draw a rectangular outline of the view. The underline and outline are drawn in the color returned by **keyboard_navigation_color()**. Using this color lends consistency to the user interface.

A BView learns that the focus has changed when its **MakeFocus()** hook function is called. It's up to **MakeFocus()** to ensure that the focus indicator is drawn or erased, depending on the BView's new status. It's usually simplest for **MakeFocus()** to call **Draw()** and have it do the work. For example:

```
void MyView::MakeFocus(bool focused)
{
   if ( focused != IsFocus() ) {
      baseClass::MakeFocus(focused);
      Draw(Bounds());
      Flush();
      . . .
   }
}
```

The BControl class has a **MakeFocus()** function that calls **Draw()**, so if your class derives from BControl, all you need to do is implement **Draw()**. **Draw()** can call **IsFocus()** to test the BView's current status. Here's a rough example:

```
void MyView::Draw(BRect updateRect)
{
```

```
   rgb_color navigationColor = keyboard_navigation_color();
   BRect r = Bounds()
   r.InsetBy(2.0, 2.0)
   . . .
   rgb_color c = HighColor();
   if ( IsFocus() ) {
      /* draw the indicator */
      SetHighColor(navigationColor);
      StrokeRect(r);
      SetHighColor(c);
   }
   else {
      /* erase the indicator */
      SetHighColor(ViewColor());
      StrokeRect(r);
      SetHighColor(c);
   }
   . . .
}
```

Handling Keyboard Actions

Finally, your BView may need to override **KeyDown()** to handle the keystrokes that are used to operate the view (for view-internal navigation). Always incorporate the inherited version of **KeyDown()** so that it can take care of navigation between views. For example:

```
void MyView::KeyDown(const char *bytes, int32 numBytes)
{
   switch ( bytes[0] ) {
      case B_ENTER:
      case B_SPACE:
            /* take action */
            break;
      case B_UP_ARROW:
      case B_DOWN_ARROW:
      case B_RIGHT_ARROW:
      case B_LEFT_ARROW:
            /* move within the view */
            break;
      default:
            baseClass::KeyDown(bytes, numBytes);
            break;
   }
}
```

Again, the BControl class implements a **KeyDown()** function that invokes the control device when the user presses the space bar or Enter key. If your class derives from BControl and it

doesn't have to do any other view-internal navigation, the BControl function may be adequate for your needs.

Drag and Drop

The BView class supports a drag-and-drop user interface. The user can transfer a parcel of information from one place to another by dragging an image from a source view and dropping it on a destination view—perhaps a view in a different window in a different application.

A source BView initiates dragging by calling **DragMessage()** from within its **MouseDown()** function. The BView bundles all information relevant to the dragging session into a BMessage object and passes it to **DragMessage()**. It also passes an image or a rectangle to represent the data package on-screen. For example:

```
void MyView::MouseDown(BPoint point)
{
   . . .
   if ( aRect.Contains(point) ) {
      BMessage message(SOME_WORDS_OF_ENCOURAGEMENT);
      message.AddString("words", theEncouragingWords);
      DragMessage(&message, aRect);
   }
   . . .
}
```

The Application Server then takes charge of the BMessage object and animates the image as the user drags it on-screen. As the image moves across the screen, the views it passes over are informed with **MouseMoved()** function calls. These notifications give views a chance to show the user whether or not they're willing to accept the message being dragged. When the user releases the mouse button, dropping the dragged message, the message is delivered to the BWindow and targeted to the destination BView.

A BView is notified that a message has arrived by a **MessageReceived()** function call. This is the same function that announces the arrival of other messages. By calling **WasDropped()**, you can ask the message whether it was dropped on the view or delivered in some other way. If it was dropped, you can find out where by calling **DropPoint()**. For example:

```
void AnotherView::MessageReceived(BMessage *message)
{
   switch ( Message->what ) {
      . . .
      case SOME_WORDS_OF_ENCOURAGEMENT:
      {
            char *words;
            if ( message->FindString("words", &words) != B_OK )
               return;
            if ( message->WasDropped() ) {
```

```
                BPoint where = DropPoint();
                ConvertFromScreen(&where);
                PleaseInsertTheseWords(words, where);
            }
            break;
        }
        . . .
        default:
            baseClass::MessageReceived(message);
    }
}
```

Aside from creating a BMessage object and passing it to **DragMessage()**, or implementing **MouseMoved()** and **MessageReceived()** functions to handle any messages that come its way, there's nothing an application needs to do to support a drag-and-drop user interface. The bulk of the work is done by the Application Server and Interface Kit.

Character Encoding

The BeOS encodes characters using the UTF-8 transformation of Unicode character values. Unicode is a standard encoding scheme for all the major scripts of the world—including, among others, extended Latin, Cyrillic, Greek, Devanagiri, Telugu, Hebrew, Arabic, Tibetan, and the various character sets used by Chinese, Japanese, and Korean. It assigns a unique and unambiguous 16-bit value to each character, making it possible for characters from various languages to co-exist in the same document. Unicode makes it simpler to write language-aware software (though it doesn't solve all the problems). It also makes a wide variety of symbols available to an application, even if it's not concerned with covering more than one language.

Unicode's one disadvantage is that all characters have a width of 16 bits. Although 16 bits are necessary for a universal encoding system and a fixed width for all characters is important for the standard, there are many contexts in which byte-sized characters would be easier to work with and take up less memory (besides being more familiar and backwards compatible with existing code). UTF-8 is designed to address this problem.

UTF-8

UTF-8 stands for "UCS Transformation Format, 8-bit form" (and UCS stands for "Universal Multiple-Octet Character Set," another name for Unicode). UTF-8 transforms 16-bit Unicode values into a variable number of 8-bit units. It takes advantage of the fact that for values equal to or less than 0x007f, the Unicode character set matches the 7-bit ASCII character set—in other words, Unicode adopts the ASCII standard, but encodes each character in 16 bits. UTF-8 strips ASCII values back to 8 bits and uses two or three bytes to encode Unicode values over 0x007f.

The high bit of each UTF-8 byte indicates the role it plays in the encoding:

- If the high bit is 0, the byte stands alone and encodes an ASCII value.
- If the high bit is 1, the byte is part of a multiple-byte character representation.

In addition, the first byte of a multibyte character indicates how many bytes are in the encoding: The number of high bits that are set to 1 (before a bit is 0) is the number of bytes it takes to represent the character. Therefore, the first byte of a multibyte character will always have at least two high bits set. The other bytes in a multibyte encoding have just one high bit set.

To illustrate, a character encoded in one UTF-8 byte will look like this (where a '1' or a '0' indicates a control bit specified by the standard and an 'x' is a bit that contributes to the character value):

```
0xxxxxxx
```

A character encoded in two bytes has the following arrangement of bits:

```
```

```
110xxxxx 10xxxxxx
```

And a character encoded in three bytes is laid out as follows:

```
1110xxxx 10xxxxxx 10xxxxxx
```

Note that any 16-bit value can be encoded in three UTF-8 bytes. However, UTF-8 discards leading zeroes and always uses the fewest possible number of bytes—so it can encode Unicode values less than 0x0080 in a single byte and values less than 0x0800 in two bytes.

In addition to the codings illustrated above, UTF-8 takes four bytes to translate a Unicode *surrogate pair*—two conjoined 16-bit values that together encode a character that's not part of the standard. Surrogates are extremely rare.

ASCII Compatibility

The UTF-8 encoding scheme has several advantages:

- The single byte that encodes an ASCII value can't be confused with a byte that's part of a multiple-byte encoding. You can test a UTF-8 byte for an ASCII value without considering surrounding bytes; if there's a match, you can be sure the byte is the ASCII character. UTF-8 is fully compatible with ASCII.
- The first (or only) byte of a character can't be confused with a byte inside a multibyte sequence. It's simple to find where a character begins. For example, this macro will do it:

```
#define BEGINS_CHAR(byte) ((byte & 0xc0) != 0x80)
```

- The string functions in the standard C library—for example, `strcat()` and `strlen()`—can operate on a UTF-8 string.
- However, it's important to remember that `strlen()` measures the string in bytes, not characters. Some Interface Kit functions, like **`GetEscapements()`** in the BFont class, ask for a character count; `strlen()` can't provide the answer. Instead, you need to do something like this to count the characters in a string:

```
int32 count = 0;
while ( *p != '0' ) {
   if ( BEGINS_CHAR(*p) )
      count++;
   p++;
}
```

- UTF-8 preserves the numerical ordering of Unicode character values. String comparison functions—such as `strcasecmp()`—will put UTF-8 strings in the correct order.
- However, you should be careful when using the string comparison functions to order a set of UTF-8 strings. Unicode tries for a universal encoding and orders characters in a way

that's generically correct, but it may not be correct for specific characters in specific languages. (Because it follows ASCII, UTF-8 is correct for English.)

- For European languages, UTF-8 generally yields more compact data representations than would Unicode. Most of the characters in a string can be encoded in a single byte. In many other cases, UTF-8 is no less compact than Unicode.

UTF-8 and the BeOS

The BeOS assumes UTF-8 encoding in most cases. For example, a **B_KEY_DOWN** message reports the character that's mapped to the key the user pressed as a UTF-8 value. That value is then passed as a string to **KeyDown()** along with the byte count:

- virtual void KeyDown(const char **bytes*, int32 *numBytes*);

You can expect the *bytes* string to always contain at least one byte. And, of course, you can test it for an ASCII value without caring whether it's UTF-8:

```
if ( bytes[0] == B_TAB )
   . . .
```

Similarly, objects that display text in the user interface—such as window titles and button labels—expect to be passed UTF-8 encoded strings, and hand you a UTF-8 string if you ask for the title or label. These objects display text using a system font—either the system plain font (**be_plain_font**) or the bold font (**be_bold_font**). The BFont class allows other character encodings, which you may need to use in limited circumstances from time to time, but the system fonts are constrained to UTF-8 (**B_UNICODE_UTF8** encoding). The FontPanel preferences application doesn't permit users to change the encoding of a system font.

Unicode and UTF-8 are documented in **The Unicode Standard, Version 2.0**, published by Addison-Wesley. See that book for complete information on Unicode and for a description of how UTF-8 encodes surrogate pairs.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSlider: Customizing your Slider

Derived from: public BControl
Declared in: be/interface/Slider.h
Library: libbe.so
Summary: *more...*

Creating a basic BSlider is very simple:

```
r.Set(5,5,255,55);
slider = new BSlider(r, "const:slider1", "Construct Speed (%)", NULL,
                0, 140);
```

This creates a slider named "const:slider1" with the label "Construct Speed (%)" above it. The range of possible values for the slider is 0 to 140.

You can add a splash of color to the slider, too. For example, if you want the slider bar to be light blue, you might add:

```
slider->SetBarColor(color);
```

You can also add hash marks to the slider:

```
slider->SetHashMarks(B_HASH_MARKS_BOTTOM);
slider->SetHashMarkCount(10);
```

In this example, **SetHashMarks()** is used to specify that the hash marks should be below the slider; you can also specify **B_HASH_MARKS_TOP** to put them above the slider, **B_HASH_MARKS_BOTH** to put them both above and below the slider, and **B_HASH_MARKS_NONE** to omit them entirely (which is the default).

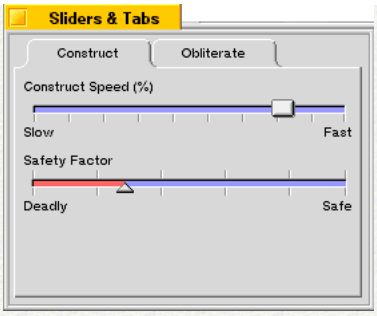
SetHashMarkCount() is called to indicate that you want 10 hash marks spaced evenly across the slider.

Finally, if you want to add labels at the ends of the slider, to indicate the minimum and maximum values, you can use the **SetLimitLabels()** function:

```
slider->SetLimitLabels("Slow", "Fast");
```

This sets the label for the minimum value (the left end of the slider) to "Slow" and the label for the maximum to "Fast".

The result is the "Construct Speed (%)" slider in the window pictures below. An example of a slider with the **B_TRIANGLE_THUMB** *thumbType* and hash marks above and below the slider is also shown.


Sliders & Tabs
Construct
Obliterate
Construct Speed (%)
Slow
Fast
Safety Factor
Deadly
Safe

Customizing through Virtual Functions

You can customize the appearance of your BSlider by overriding virtual functions in the BSlider class that are responsible for drawing the slider bar, thumb, focus mark, and hash marks.

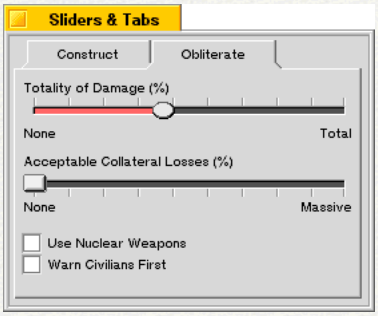
For example, let's say you want your slider's thumb to be round. Just create a new class—let's call it CustomSlider—derived from BSlider, and override the **DrawThumb()** function with code that might look something like this:

```
const rgb_color kWhite = {255,255,255,255};
const rgb_color kWhiteGray = {235,235,235,255);
const rgb_color kDarkGray = {100,100,100,255};
const rgb_color kBlackColor = {0,0,0,255};

void CustomSlider::DrawThumb(void) {
   BRect r;
   BView *v;

   // Get the frame rectangle of the thumb
   // and the offscreen view.

   r = ThumbFrame();
   v = OffscreenView();

   // Draw the black shadow

   v->SetHighColor(kBlackColor);
   r.top++;
   r.left++;
   v->StrokeEllipse(r);

   // Draw the dark grey edge

   v->SetHighColor(kDarkGray);
   r.bottom--;
   r.right--;
   v->StrokeEllipse(r);

   // Fill the inside of the thumb

   v->SetHighColor(kWhiteGrayColor);
   r.InsetBy(1,1);
   v->FillEllipse(r);
}
```

All rendering of a BSlider is done into an offscreen view, which you can get a pointer to by calling the `OffscreenView()` function. This improves performance and makes drawing the slider—especially complicated ones—look much smoother to the user.

This code gets a pointer to the offscreen view, then renders a round, beveled thumb into the offscreen view, filling the thumb's frame rectangle as returned by the `ThumbFrame()` function.

The result is a round thumb that looks like the one in the "Totality of Damage" slider in the picture below:



The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Interface Kit

Most applications have an interactive graphic user interface. When an app starts, it displays a set of windows in which the user can click and type. The application responds to the user's actions, and updates its window to show the user that it's listening.

To run this kind of user interface, an application has to:

- Manage a set of windows
- Draw within the windows
- Respond to the users's actions (reported as *interface messages*)

The Interface Kit defines a set of C++ classes that provide a structure for these operations. This chapter first introduces the conceptual framework for the user interface, then describes all the classes, functions, types, and constants the kit defines.

Framework for the User Interface

A graphical user interface is organized around windows. In a multitasking environment, any number of applications might be running at the same time, each with its own set of windows on-screen. The windows of all running applications must cooperate in a common interface. The Application Server manages this mess. It's the conduit for an application's message input and drawing output:

- It monitors the keyboard and mouse and sends messages reporting each user keystroke and mouse action to the application.
- It receives drawing instructions from the application and interprets them to render images within windows.

BWindow Objects

Every window in an application is represented by a separate BWindow object. Constructing the BWindow establishes a connection to the Application Server. When you call BWindow's window-manipulating functions (`Show()`, `MoveTo()`, `SetTitle()` and so on), the object sends a message to the server, which performs the actual manipulation.

The BWindow class inherits from BLooper. Every BWindow object spawns a thread (in the application's address space) where it receives and responds to interface messages from the server.

All other Interface Kit objects play roles that depend on a BWindow. They draw in a window, respond to interface messages received by a window, or act in support of other objects that draw and respond to messages.

BView Objects

A window is divided into smaller rectangular areas called *views*. Each view corresponds to one part of what the window displays—a scroll bar, a document, a list, and so on.

An application sets up a view by constructing a BView object and associating it with a particular BWindow. The BView object is responsible for drawing within the view rectangle and for handling interface messages directed at that area.

Drawing in a View

A window can retain and display rendered images, but it can't draw them; for that it needs a set of BViews. A BView is an agent for drawing, but it can't render the images it creates; for that it needs a BWindow. The two objects work hand in hand.

Each BView object is an autonomous graphics environment for drawing: It has its own coordinate system, current colors, drawing mode, clipping region, font, pen position, and so on. The BView class also defines functions that represent elemental drawing operations such as line stroking, character drawing, and image blitting.

Handling Messages in a View

When the user acts, system messages that report the resulting events are sent to the BWindow object, which determines which BView elicited the user action and should respond to it. For example, a BView that draws typed text can expect to respond to messages reporting the user's keystrokes. A BView that draws a button gets to handle the messages that are generated when the button is clicked. The BView class derives from BHandler, so BView objects are eligible to handle messages dispatched by the BWindow.

The View Hierarchy

A window typically contains a number of different views—all arranged in a hierarchy beneath the *top view*, a view that's exactly the same size as the content area of the window. The top view is a companion of the window; it's created by the BWindow object when the BWindow is constructed. When the window is resized, the top view is resized to match. Unlike other views, the top view doesn't draw or respond to messages; it serves merely to connect the window to the views that the application creates and places in the hierarchy.

The view hierarchy can be represented as a branching tree structure with the top view at its root. All views in the hierarchy (except the top view) have one, and only one, parent view. Each view (including the top view) can have any number of child views.

When a new BView object is created, it isn't attached to a window and it has no parent. It's added to a window by making it a child of a view already in the view hierarchy. This

is done with the `AddChild()` function. A view can be made a child of the window's top view by calling BWindow's version of `AddChild()`.

Until it's assigned to a window, a BView can't draw and won't receive reports of events. BViews know how to produce images, but it takes a window to display and retain the images they create.

Drawing and Message Handling in the View Hierarchy

The view hierarchy determines what's displayed where on-screen, and also how user actions are associated with the responsible BView object:

- When the views in a window are called on to draw, parents draw before their children; children draw in front of their ancestors.
- Mouse events are associated with the view where the cursor is located.

Overlapping Siblings

Siblings don't draw in a predefined order. If they overlap, it's indeterminate which view will draw last—that is, which one will draw in front of the others. Similarly, it's indeterminate which view will receive the mouse events that occur in the area the siblings share.

Therefore, it's strongly recommended that you arrange your sibling views so they don't overlap.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBitmap Function Summary

Function Summary

Function	Description
AddChild()	Add a view to a BBitmap
Archive()	Archives the BBitmap into a BMessage
BBitmap()	Constructor
~BBitmap()	Destructor
Bits()	Returns pointer to bitmap data
BitsLength()	Returns length of bitmap data in bytes
Bounds()	Returns bounds rectangle of the BBitmap
BytesPerRow()	Returns number of bytes required to store one line of data
ChildAt()	Returns pointer to a child view
ColorSpace()	Returns color space of bitmap data
CountChildren()	Returns number of child views
FindView()	Returns pointer to a child view
Instantiate()	Creates a BBitmap from a BMessage
IsLocked()	Tests whether the off-screen window is locked
IsValid()	Returns true if memory exists for the bitmap
Lock()	Locks off-screen window
RemoveChild()	Removes a child view
SetBits()	Imports bitmap data
Unlock()	Unlocks off-screen window

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSmallBuffer

Derived from: BBuffer
Declared in: be/media/Buffer.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BSmallBuffer class provides small, efficient buffers for MIDI and other small packet-based media types.

To use BSmallBuffers, create a BSmallBuffer, then fill it with up to `SmallBufferSizeLimit()` bytes of data. Then call `SendBuffer()`. You can then immediately reuse the buffer.

How is this possible? BSmallBuffers don't go in buffer groups. The receiver immediately either processes the buffer (if it's time) or caches it until it's needed, then recycles the buffer.

If you're writing a node, and receive a buffer with the **`B_SMALL_BUFFER`** flag set, you must recycle the buffer in your `BufferReceived()` function.

Constructor

BSmallBuffer()

```
BSmallBuffer(void)
```

Constructor. Creates a small buffer that can hold up to `SmallBufferSizeLimit()` bytes.

Static Functions

SmallBufferSizeLimit()

```
size_t *SmallBufferSizeLimit(void)
```

Returns the maximum number of bytes a BSmallBuffer can contain.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSmallBuffer Function Summary

Function Summary

Function	Description
BSmallBuffer()	Constructor
SmallBufferSizeLimit()	Returns the maximum number of bytes that can be put into a small buffer

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Media Kit

The Media Kit provides powerful support for all forms of media (including, but not limited to, audio and video), including both playback and recording from and to a wide variety of media and devices.

There are two levels of Media Kit programming: the high level, where an application accesses the Media Kit to play and record sound and video, and the low level, which involves actually creating the nodes that manipulate media data.

Media Kit Concepts

This section is a general overview of key Media Kit concepts.

Nodes

The first thing that you need to understand as a media programmer working with the BeOS is the concept of a **node**. Generically speaking, a node is a specialized object in the media system that processes buffers of media data. All nodes are indirectly derived from the BMediaNode class (but never directly from BMediaNode; instead, nodes are derived from specialized node type classes).

Nodes can be loaded from add-on modules, or they can be created within an application itself. See the BMediaAddOn class for details.

The **node kind** (defined by the `node_kind` type) is a description of the basic capabilities of a node. See "Types of Nodes" on page4.

A `media_node` is a structure that an application uses when working with media nodes; interactions with the Media Roster are almost always done using this structure instead of an object actually derived from BMediaNode. The reason for this that applications will often need to share nodes, and due to protected memory, it's not really feasible to pass a BMediaNode pointer among the applications.

No portion of the media node protocol is optional; if you don't follow all the rules precisely, you risk hurting media performance, and since BeOS is the Media OS, that would be a bad thing.

For more detailed information about the architecture of the Media Kit (in particular, how nodes relate to one another), please see the BMediaNode class.

Types of Nodes

There are several basic kinds of nodes. Each of them is derived originally from the BMediaNode class. Any nodes that you might implement will be derived, in turn, from one or more of these node types. The **node kind** indicates which of these types a node implements.

Producers

A producer (a node derived from the BBufferProducer class) outputs media buffers, which are then received by consumers. A producer might be generating nodes on its own (for example, a tone generator might be using a mathematical formula to generate a sound, or an audio file player might be loading data from disk and sending buffers containing its audio data). Other producers might be responsible for acquiring data from media hardware (such as a video camera) and passing the media buffers to consumers down the line.

In the example in the "Communicating With Nodes" section, the sound file reader node is an example of a producer.

Consumers

Consumers (nodes derived from BBufferConsumer), receive buffers from a producer and process them in some manner. For example, a sound card's software would provide a consumer node that receives audio buffers and plays them through the card's hardware.

In the example in the "Communicating With Nodes" section, the sound player node is an example of a consumer.

Consumer/Producers (Filters)

A consumer/producer (a node that derives from both BBufferConsumer and BBufferProducer) is also called a **filter**. A filter accepts buffers (like a consumer), processes them in some manner, then sends them back out again (like a producer). This can be used to alter sound or video data.

For example, an audio filter might add a reverb effect to sound buffers, or a video filter might add captioning to a video stream.

In the example in the "Communicating With Nodes" section, the equalizer node is an example of a filter node.

Controllables

If a node wishes to provide the user options for configuring its functionality, the node can derive from BControllable. This provides features for creating a network of controllable parameters, and for publishing this information to Media Kit-savvy applications (including the Media preference applications).

Time Sources

A time source node broadcasts timing information that can be used by other nodes. All nodes are slaved to a time source, which provides synchronization among all nodes slaved to that time source. Typically, applications won't need to worry about this, because any node created through the BMediaRoster class are automatically slaved to the system (default) time source.

A node can be derived from any of these types (BBufferProducer, BBufferConsumer, BTimeSource, and BControllable), in any combination, as appropriate.

The Node Kind

The **node_kind** type is used to identify which of these interfaces a node implements; this lets the Media Kit know which APIs your node supports. These flags include **B_BUFFER_PRODUCER**, **B_BUFFER_CONSUMER**, **B_TIME_SOURCE**, and **B_FILE_INTERFACE**.

There are other flags available in the node kind; these indicate special node functions supported by the node. These are **B_PHYSICAL_INPUT**, which indicates that the node is a physical input device, **B_PHYSICAL_OUTPUT**, which indicates that the device is a physical output device, and **B_SYSTEM_MIXER**, which indicates that the node is the system mixer.

The primary purpose to these flags is to help user interfaces determine how to draw the interface for these nodes; they get special icons in the Media preference application, for example.

Deriving From Multiple Classes

For example, if you're creating a sound card node that plays audio to stereo speakers, you would need to derive from BBufferConsumer (in order to receive audio buffers). You could also derive from BTimeSource if your sound card has the ability to provide timing information to others. And if you want the user to be able to control the volume, balance, and so forth, you would also derive from BControllable.

If your sound card also provides a digitizer input, you would actually create a second node to support that feature. It would inherit from BBufferProducer (so it can generate audio buffers for other nodes to use). It might also derive from BTimeSource and BControllable.

But not all nodes necessarily represent a physical hardware device. If you want to create a filter—for example, a noise-reduction filter—you can create a node to do this too. Simply derive from both BBufferConsumer (so you can receive buffers) and BBufferProducer (so you can send back out the altered buffers).

Source & Destination vs. Output & Input

Beginning Media Kit programmers may have trouble understanding the difference between a **media_source** and a **media_output**, and a **media_destination** and a **media_input**.

The **media_source** and **media_destination** structures describe a "socket" of sorts (much like in networking). These are the ends of the connection, much like the jacks you might plug cables into to connect various components of a stereo system. They're relatively small, lightweight descriptions containing only the information needed during real-time manipulation of nodes. Buffers travel from the source to the destination. You can use the basic operators (=, ==, and !=) on these objects.

media_source::null and **media_destination::null** represent uninitialized endpoints.

The **media_output** and **media_input** structures describe an actual connection between a **media_source** and a **media_destination**, including the source and destination, the connetion's name, and the format of the data the connection is intended to handle. These are larger, and contain additional information needed when presenting a user interface describing the connections between nodes.

The **media_input** structure describes the receiving (upstream) end of a connection; **media_output** describes the sender (the downstream end).

Although **media_output** and **media_input** contain all the information of the **media_source** and **media_destination** structures, the latter structures exist because when you're doing real-time manipulation of media data, you don't want to be tossing large blocks of data around unless you have to. And you don't have to.

Media Formats

The media_format structure describes the type of media being passed through a connection. The application and the nodes with which it's working negotiate the format through a sequence of calls and callbacks.

This structure contains a basic media type (such as **B_MEDIA_RAW_AUDIO** or **B_MEDIA_ENCODED_VIDEO**) and a union containing additional information depending on the basic type.

Raw Audio

The **media_format** structure describing **B_MEDIA_RAW_AUDIO** media contains the following information:

- Frame rate: the number of frames of audio per second.
- Format: the data type of each audio sample.
- Channel count: the number of samples per frame. For example, stereo sound has two channels (left and right).
- Byte order: indicates whether the data is big-endian or little-endian.
- Buffer size: the number of bytes of data per buffer.

Raw Video

The **media_format** structure describing **B_MEDIA_RAW_VIDEO** media contains the following information:

- Field rate: the number of fields (buffers) of video per second.
- Interlace: the number of fields per frame.
- Display information: the color space, line width, line count, bytes per row, and so forth.

Format Wildcards

A format wildcard indicates unspecified parts of the format. This is used during negotiation so a node or application can say "I don't care about this partifular part of the format; I'm flexible." For example, if your application is negotiating with a video node, and you can handle any bit depth of video, you would specify a wildcard in the color space field of the format.

Each format type has one wildcard object.

For example, if you're preparing to negotiate with an audio node, and don't care about the frame rate, the following code will set up the **media_format** structure for negotiation:

```
media_format format;
media_raw_audio_format wc;

wc = media_raw_audio_format::wildcard;
```

```
format.type = B_MEDIA_RAW_AUDIO;
format.u.raw_audio.frame_rate = wc.frame_rate;
```

Audio: Samples and Frames and Buffers (Oh my)

A *sample* is a single value that defines the amplitude of an audio waveform at a given instant in time. This value can be represented in a number of ways: as a one-byte integer, as a two-byte integer, as a floating-point value, or in other ways. The native audio sample format in BeOS is floating-point, where a value of 0.0 represents a wave point with no amplitude. Positive values indicate a sample whose amplitude is above the zero point, and negative values represent samples below the zero point.

A *frame* is the set of samples that describes a sound at a given instant in time. If the audio contains multiple channels, a frame contains multiple samples, one for each channel. A monaural audio stream contains one sample per frame, while a stereo audio stream contains two samples per frame. Surround sound formats contain even more samples per frame of audio.

A *buffer* is a more efficient way of sending audio from one node to another. Instead of beaming around thousands of individual frames, frames are grouped into buffers that are then transferred *en masse* along the chain of nodes. This improves throughput and reduces overhead.

See "media_raw_audio_format" on page343 for more information about describing audio formats.

Video: Interlacing, Fields, and Frames

Video can be represented either as non-interlaced or interlaced data. Both NTSC video (the standard format for television video in the United States) and PAL video (the standard in many other countries) are interlaced.

Typically, a video stream will have either one or two fields per frame. If the video isn't interlaced, there's only one field, which contains all the scan lines in the video stream. If the video is interlaced, there are usually two fields per frame (it might be more, but two is most common). One field contains the even lines of the frame, the other contains the odd lines.

Video buffers in the BeOS Media Kit always contain either a complete frame or a complete field; buffers never contain partial frames or fields.

See "media_raw_video_format" on page344 for more information about describing the format of video data.

Reference Material

This chapter doesn't pretend to be a tutorial on the intricacies of audio and video data formats. There are plenty of good reference books on these subjects. Here's a list of books our engineers suggest:

- **The Art of Digital Audio** by John Watkinson (ISBN 0240513207)
- **Compression in Audio and Video** by John Watkinson (ISBN 0240513940)
- **Digital Audio Signal Processing** by Udo Zolzer (ISBN 0471972266)
- **Introduction to Signal Processing** by Sophocles Orphanidis (ISBN)
- **Principles of Digital Audio** by Ken C. Pohlmann (ISBN 0070504695)
- **A Programmer's Guide to Sound** by Tim Kientzle (ISBN 0201419726)
- **Video Demystified** by Keith Jack (ISBN 18780723X)

Buffers

A BBuffer is a packet of media data that can be passed between nodes. It contains a header that describes the data and the actual data being transmitted.

The buffer header contains inforrmation that tells you what the data is:

- Start time: the time at which the data should be performed.
- Size used: the number of valid data bytes in the buffer.
- Additional information based on the media type.

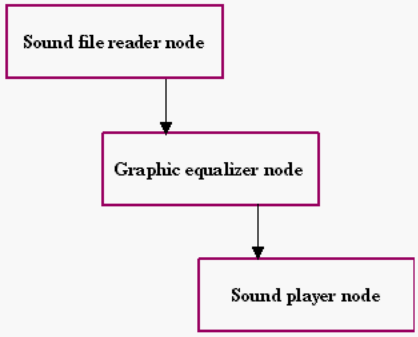
For optimal performance, the media system stores buffer data in shared memory areas so that different applications and nodes can access the same buffer, without having to copy it around in memory. Buffers are passed from one node to the next by giving each node a BBuffer that references the same area of memory.

Communicating With Nodes

The BMediaRoster class provides the application interface to all available nodes (whether they're created by the application or by an add-on). An application instantiates the nodes it needs, then establishes the connections between them that will accomplish the desired task.

For example, if an application wants to play a sound file through a graphic equalizer, it might first instantiate a node that reads sound data from a disk file and outputs audio buffers, then instantiate a node that performs filtering on audio buffers, and finally, a node that plays sound buffers to the speakers.

Once these three nodes are instantiated, the application creates the links between them. The output of the audio file reading node is connected to the input of the equalizer node, then the equalizer node's output is connected to the sound player node's input.:



Once these connections are established, the application can then begin playing the sound file, by telling the first node what sound file to play, and then starting all of the nodes running. The sound file reader creates buffers containing audio data, and passes those along to the equalizer node, which alters them, then passes them along to the sound player node, which plays the buffers and recycles them for reuse. A more detailed example of how to work with nodes to play back media data is given in the BMediaRoster class.

Connections at the Low Level

Each media node maintains a **control port**. The Media Kit interacts with the node by sending messages to the node's control port.

Time

One of the most important issues in manipulating media data is to properly track and synchronize precisely-timed events. There are several kinds of time (and you don't even have to be Stephen Hawking to understand them):

Media Time

Media time is time relative to a particular media file, and is represented in terms of microseconds since the beginning of the file. Seek operations are performed using media time.

Real Time

Real time is the time reported by the system clock, and is represented in terms of microseconds since the computer was booted. It's used in conjunction with system calls, such as `snooze()`, `read_port_etc()`, and so forth.

Performance Time

Performance time is the time at which events should occur at the final output. It's reported by a time source node, which establishes the relationship between real time and performance time.

Usually a single time source is the master clock for all the nodes in a node chain.

Performance Time vs. Real Time

There are two reasons why performance time and real time may differ: if the master clock isn't the system time, there may be drift over time. Also, latency caused by the time it takes for data to pass from one node to another can cause drift as well.

Latency

Latency is the amount of time it takes to do something. There are four kinds of latency:

- **Algorithmic latency** is any intentional or unavoidable delay added to the processing of buffers, such as retimestamping buffers so they're performed later. This also includes delays caused, for example, by a filter node requiring that it hold samples for a while to compute delay or echo effects.
- **Processing latency** is the time it takes for a node to process a buffer.
- **Scheduling latency** is the amount of time it takes to schedule a buffer.
- **Downstream latency** is the amount of time that passes between the time a node sends a buffer to the time at which the buffer can be performed. This includes the internal latencies of every downstream node.

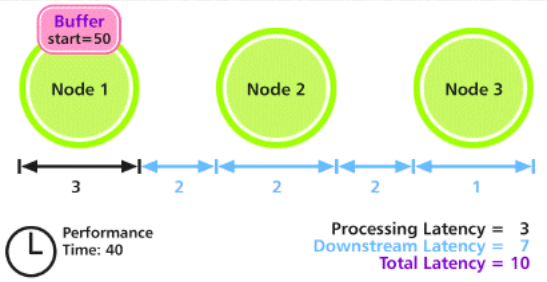
A Latency Example

Let's consider a case in which three nodes are connected. The first node has a processing latency of 3 microseconds, the second has a processing latency of 2 microseconds, and the last has a processing latency of 1 microsecond.

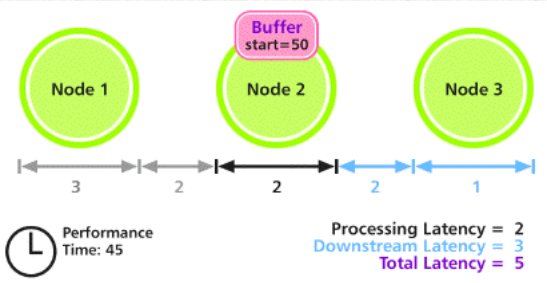
This example uses the term "microseconds" because the Media Kit measures time in microseconds; however, the latencies used in this example may not be indicative of a real system.

In addition, 2 microseconds is required for buffers to pass from one node to the next. The total latency of this chain of nodes, then, is 3 + 2 + 2 + 2 + 1 = 10 microseconds.

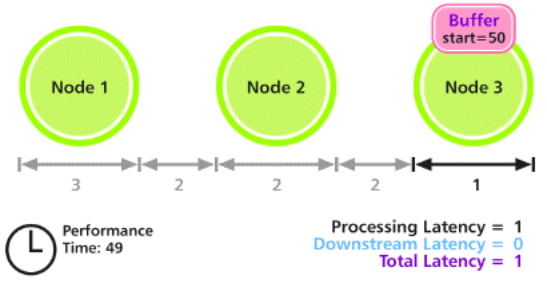
A buffer is scheduled to be played at a performance time of 50 microseconds. In order to get this buffer to the last node in time to be played at the right time, it needs to begin being processed at 40 microseconds. We see this in the diagram below.



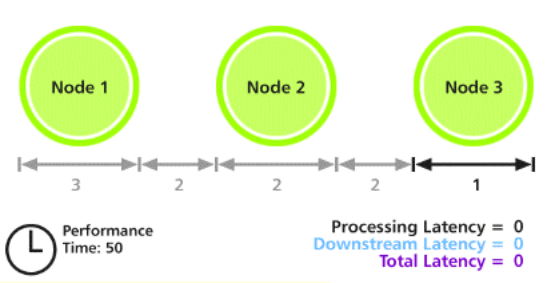
After the buffer has been processed by Node 1 and has been passed along to Node 2, 5 microseconds have passed. Node 2 will take 2 microseconds to process the buffer. This gets us to a performance time of 45 microseconds:



Node 2 processes the buffer, and passes it along to Node 3. This takes a total of 4 microseconds (2 microseconds of processing time plus 2 microseconds to be sent to Node 3). We arrive at the performance time of 49 microseconds:



Finally, Node 3 processes the buffer; this requires 1 microsecond of processing time. At this point, the performance time of 50 microseconds has been reached, and the buffer has been performed on schedule.



Using the Media Kit

If you're writing an application that wants to record or play back some form of media data (such as a sound or a video file), all your media needs are served by the BMediaRoster class. This class provides access to the various nodes, and lets you establish the relationships among them that are necessary to perform the tasks you'd like to accomplish.

The BMediaNode is an abstract class; you don't call its functions directly. Instead, you use BMediaRoster calls to issue requests to the various nodes available on the BeOS system on which your application is running. In addition, you can't derive a new class directly from BMediaNode; instead, derive from one of the system-defined subclasses (BBufferConsumer, BBufferProducer, BControllable, and so forth).

Media Kit error code constants can be found in **be/media/MediaDefs.h**.

The Media Roster

The Media Roster manages an application's communication with the media system. Each application has at most one instance of the media roster. The roster is obtained by calling `BMediaRoster::Roster()`; if it already exists, the current roster object is returned.

This section briefly summarizes some of the functions served by the Media Roster; for more detailed information, see the BMediaRoster class.

Finding the Right Nodes

There are several standard nodes, which the user configures using the Media preference application, plus the system mixer. The BMediaRoster class provides convenience routines to quickly get references to these nodes, such as `GetAudioMixer()` and `GetVideoOutput()`. See the BMediaRoster class for details.

If you need some other node, you can browse through the available nodes to find the one best-suited for your needs. Nodes are created from **dormant nodes**, which live inside media add-ons. Each dormant node is a reference to a **node flavor**, a structure that describes the nodes the dormant node can create.

Once you've identified the best node for your purposes, you negotiate and establish a connection to the node. This is discussed in the BMediaRoster overview.

Controlling Nodes

Once your nodes are created and connected to each other, you can control them by using the BMediaRoster functions `Preroll()`, `Seek()`, `Start()`, and `Stop()`. These let you move to specific points in the media file and start and stop playback or recording.

You can also set the nodes' time sources, run modes, and play rates.

Displaying a User Interface

BControllable nodes can present a user interface representing the aspects of itself that the user can configure. Each of these configurable aspects are called a **parameter**. The BMediaRoster provides functions that let you create a user interface for a node's parameters. See `BMediaRoster::StartControlPanel()` for the easiest way to do this.

Media Files

The approved way to access files containing media data is via the BMediaFile and BMediaTrack classes. If you're using a node-based playback or recording system, and you want to have easy access to media files, you can get access to the node used by the BMediaFile class by calling `BMediaRoster::SniffRef()`.

See "Reading and Writing Media Files" for an example of how to access media files the right way.

The Audio Mixer

The audio mixer accepts as input audio data which it then mixes and outputs to the audio output device or devices the user has selected in the Audio preference application. Your application can get a media_node referencing the audio mixer using the `BMediaRoster::GetAudioMixer()` function. You can't intercept audio being output by the audio mixer; they go directly to the output device.

Buffers containing any standard raw audio format can be sent to the audio mixer; the mixer will convert the data into the appropriate format for playback.

The audio mixer is always running, and is slaved to the most appropriate time source. You should never change its time source or start or stop the audio mixer (in other words, don't call the BMediaRoster calls `SetTimeSourceFor()`, `Start()`, or `Stop()` on the audio mixer).

The Audio Input

The audio input creates audio buffers from external sources, such as microphones or line in ports. The physical hardware device from which the sound is input is configured by the user using the Audio preference application.

In the current implementation of the Media Kit, the audio input doesn't let you change the sampling rate. This may change in the future. To ensure that your application will continue to work in the future, don't assume that the current sampling rate will remain in effect; instead, you should look at the `media_format` structure in the `media_output` you're using for your connection to the audio input:

```
if (input->format.media_raw_audio_format.frame_rate != MY_FRAME_RATE) {
   /* it's the wrong frame rate */
}
```

The audio input is exclusive: only one connection to it is allowed at a time. If you need to receive buffers from the input from two consumers, you'll need to create a special node that receives audio buffers, then sends copies of them to all the consumers that are attached to it.

Audio Playback Made Easy

If all you want to do is play back raw audio (such as AIFF or WAVE files), the Media Kit provides the BSoundPlayer class to simplify this process. BSoundPlayer hides the inner workings of the Media Kit from you to make your life simple. See these two classes for more information; an example on how to play audio files is given in the BSoundPlayer class overview.

You might also want to consider the various sound playback classes provided by the Game Kit, such as BSimpleGameSound and BFileGameSound.

If this is still too much for you, you can use the `play_sound()` global C function to play a sound file. The `stop_sound()` function can be used to stop a sound started using `play_sound()`, and `wait_for_sound()` lets you block until the sound finishes playing.

Creating New Node Classes

You can create your own nodes to perform different types of media processing. Nodes can be provided in add-ons that the Media Kit can load dormant nodes from, or in the application itself. This is discussed in detail in the sections on BMediaNode, BBufferConsumer, and BBufferProducer.

If your node uses multiple threads, make sure you thread-protect calls to other nodes (in particular, be sure you thread-protect calls to the BTimeSource). Use a semaphore or other appropriate protection mechanism.

As a general rule, you should use the BMediaEventLooper class to handle the low-level scheduling and queuing of media events. See "A BMediaEventLooper Example" for an example of how this is done, including an explanation of the key points of creating a new media node.

Creating a Media Add-on

This is discussed in detail in the BMediaAddOn class overview.

Application-based Nodes

You can create your own node subclasses in an application if your application has special needs; just derive from the appropriate base class (such as BBufferConsumer) as normal. Note, however, that your application should never directly call any of your subclass' functions; instead, you should register the node with the media roster, and control it via BMediaRoster calls, just like any other node, by using the media_node that describes your node.

Once you've written the code for your node class, you can register it with the Media Server by calling **`BMediaRoster::RegisterNode()`**. When you're done with the node, you need to unregister it by calling **`BMediaRoster::UnregisterNode()`**. The easiest way to do this is just have the node class unregister itself when it's deleted.

Timing Issues

When dealing with a number of nodes cooperating in processing data, there are always important timing concerns. This section covers how various types of nodes need to behave in order to maintain proper timing.

Calculating Buffer Start Times

To calculate the presentation time at which a buffer should be performed, you should keep track of how many frames have been played, then multiply that value by 1000000LL/sample_rate (and, if your calculation is being done using floating-point math, you should **`floor()`** the result). You can then apply whatever offset you want to **`Seek()`** to.

```
buf->Header()->size_used = your_buf_frames * your_frame_size;
```

```
buf->Header->start_time = your_total_frames*1000000LL/your_format.frame_rate;
your_total_frames += your_buf_frames;
```

You shouldn't compute the start time by adding the previous buffer's duration to its start time; the accumulation of rounding errors over time will cause dropped samples about three times per second if you do.

Producers

Producers that produce buffers intended for output need to stamp each buffer it creates with a *startTime*, which indicates the performance time at which the buffer should be played. If the producer is playing media from a file, or synchronizing sound, this is the time at which the media should become analog.

In order to compute this *startTime* properly, the producer must prepare the buffers in advance, by the amount of time reported by `BBufferProducer::FindLatencyFor()`. The producer also needs to respond to the `BBufferProducer::LateNoticeReceived()` hook function by at least updating the time stamps it's putting on the buffers it's sending out, so they'll be played by the downstream nodes, which should be checking those times to play them at the correct time (and may be dropping buffers if they're late). If this isn't done, things will tend to get further and further behind.

In general, it's best to try to produce buffers as late as possible without actually causing them to arrive at their destination late (ie, they should be sent at or before the time *presentationTime* - *downstreamLatency*). This will ensure the best overall performance by reducing the number of buffers that are pending (especially if the user starts playing with time such that your node gets seeked or stopped). Also, if you're producing buffers that have a real-world connection, such as to a video display, producing them too early might cause them to be displayed early.

If a producer is producing buffers that are being generated by a physical input (such as a microphone jack, for example) handle things somewhat differently. They stamp the generated buffers with the performance time at which they were captured (this should be the time at which the first sample in the buffer was taken). This means that when these buffers are transmitted downstream, they'll always be "late" in the eyes of any node they arrive at.

This also means you can't easily hook a physical input to a physical output, because buffers will always arrive at the output later than the timestamped value. You need to insert another node between the two to adjust the time stamps appropriately so they won't be "late" anymore.

You can call `BMediaRoster::SetProducerRunModeDelay()` on a physical input producer to make it appropriately retimestamp buffers it generates automatically when recording.

Additionally, nodes that record data (such as file-writing nodes), in the `B_RECORDING` run mode, shouldn't care about buffers that arrive late; this lets data be recorded without concern for this issue.

Consumers

If the consumer is the device that recognizes the media (ie, it plays the audio or video contained in the buffers it receives), it needs to report the correct latency back to the producer for the time it takes buffers to reach the analog world (ie, the amount of time it takes to present the data in the buffer to the user, whether it's audio or video). Buffers that are received shouldn't be played until the *startTime* stamped on the buffers arrives. If buffers arrive late, the consumer should send a late notice to the producer, so it can make the necessary adjustments, and not pass the buffer along at all; be sure to `Recycle()` the buffers so they can be reused.

Consumer/Producers (Filters)

A consumer/producer (filter) must report the correct latency for the time a buffer takes to pass through the filter from the time it's received to the time it's retransmitted, plus the downstream latency. It shouldn't change the time stamp, unless this explicitly part of the filter's purpose. The filter should also handle late packets as described under Producers and Consumers above.

Media Applications

The application that starts the nodes and the time source to which they're slaved needs to provide them with the correct starting times. For example, if several nodes have been connected, they've all been slaved to an appropriate time source, and you want to start them all up, you need to take the following steps:

- Find the latency of the entire network, so you can give it time to come up to speed.
- Start all the nodes with the performance time at which they should start playing.
- Seek the time source to the performance time and real time that you want the performance time to be related to. This is crucial: it establishes the relationship between performance time and real time; if you forget to do this, things will look and sound very unpleasant as the media tries desperately to adjust to the actual time.

```
bigtime_t latency;
Roster->GetLatencyFor(node1, &latency);

Roster->PrerollNode(node1);
Roster->PrerollNode(node2);
Roster->PrerollNode(node3);
Roster->StartNode(node1, 0);
Roster->StartNode(node2, 0);
Roster->StartNode(node3, 0);

bigtime_t now = system_time();
Roster->SeekNode(timesourceNode, -latency, now + 10000);
Roster->StartNode(timesourceNode, now + 10000);
```

The extra 10,000 microseconds is added in case the code gets preempted while preparing to start the timesourceNode; this gives us a little fudge factor so we don't start out behind.

B_OFFLINE Run Mode

Nodes that run in offline mode are a special case in the timing world.

Consumers

in `B_OFFLINE` mode derive their current time from the arrival of buffers on their inputs. The current performance time is the minumum of all times received on its active inputs. Active inputs are those inputs that are connected and haven't received a `ProducerDataStatus()` call indicating that there are no buffers coming. You receive a time when you get a buffer `start_time` or a `ProducerDataStatus()` call with **`B_PRODUCER_STOPPED`**.

Consumers in offline mode should call `RequestAdditionalBuffer()` once it's received and processed a buffer on one of its inputs in order to obtain further buffers.

Producers

just send buffers in sequence in `B_OFFLINE` mode. The recommended behavior is to send the first buffer, then wait for an `AdditionalBufferRequested()` call before sending the next buffer. If this request doesn't arrive within a reasonable amount of time (a second or so, depending on your application), your node should accept that it's working with a not-so-bright consumer and start sending buffers at your convenience.

Don't call the time source from your producer in `B_OFFLINE` mode.

If a producer has ever received an `AdditionalBufferRequested()` call, it should assume that the consumer knows what it's doing and should only send buffers on request.

Installing Media Nodes and Drivers

Media node add-ons should be installed in the **/boot/home/config/add-ons/media** directory.

Media drivers should be installed in **/boot/home/config/add-ons/kernel/drivers/bin**. Then create a simlink to the driver in **/boot/home/config/add-ons/kernel/drivers/dev/(type)**, where (type) is the type of driver you're installing (audio, video, etc).

After installing a media node add-on, you have to restart the Media Server for it to become available for use.

About enum Members of Classes

The Media Kit has several classes (most notably, BMediaNode) that contain, as members, enums. For instance, in BMediaNode, you'll find the following:

```
class BMediaNode {
   ...
   enum run_mode {
      B_OFFLINE = 1,
      B_DECREASE_PRECISION,
      B_INCREASE_LATENCY,
      B_DROP_DATA,
      B_RECORDING
   };
   ...
};
```

In this case, you can freely use **B_OFFLINE** and so forth from within objects derived from BMediaNode, but if you want to use these values from other classes (or outside any class), you need to use the notation **BMediaNode::B_OFFLINE** to use these constants. This is true of any enum defined within a class; this will be called-out specifically in the descriptions of any constants in this chapter.

About Multiple Virtual Inheritance

Virtual inheritance is slightly different from regular inheritance in C++. The constructor for the virtual base class has to be explicitly (or implicitly) called from the most-derived class being instantiated, rather than being called from the direct descendant class actually defining the virtual inheritance.

In simple terms, this means that whenever you derive a new class from a class that uses virtual inheritance, your derived class's constructor should explicitly call the parent class's constructor.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BControllable

Derived from: virtual BMediaNode
Declared in: be/media/Controllable.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A node whose behavior can be controlled by the user should be derived from BControllable, as well as from whatever other interface classes the node might be derived from. Deriving from BControllable lets the node publish information about the parameters that can be adjusted and how they relate to each other.

A client application can use the published information to build a user interface for the node or pass the information through to a system routine that will build the interface.

A node can also have the ability to start its own control panel under outside control, which can take advantage of special knowledge of the node.

Hook Functions

- `GetParameterValue()`
- `HandleMessage()`
- `SetParameterValue()`
- `StartControlPanel()`

Constructor

BControllable()

protected:

BControllable()

The BControllable constructor. You should override this in your derived class to create a BParameterWeb object, configure it to describe the available parameters, and call **BControllable::SetParameterWeb()** with that object before returning.

Member Functions

ApplyParameterData() see MakeParameterData()

BroadcastChangedParameter()

protected:

```
status_t BroadcastChangedParameter(int32 parameterID)
```

When the configuration of a specific parameter changes, you should call this function with the ID of the changed parameter so that clients know that they need to check with the node to determine the parameter's new configuration.

The configuration of a parameter changes only when the range of possible values for the parameter changes. For example, if the parameter's value is a CD track number, the configuration would change (thus requiring a call to `BroadcastChangedParameter()`) if the user put in a different CD with a different number of tracks on it.

RETURN CODES

- B_OK. No errors.
- `BMessenger::SendMessage()` errors.

BroadcastNewParameterValue()

protected:

```
status_t BroadcastNewParameterValue(bigtime_t when,
    int32 id, void *newValue, size_t valueSize)
```

Call this function when a parameter value change takes effect and you want people that are interested in knowing about the change to stay in sync with you. Unlike `BroadcastChangedParameter()`, this function actually passes along the new value of the parameter.

The *when* argument indicates the performance time at which the change took effect. The *id* indicates the parameter ID of the parameter whose value changed. *newValue* is a pointer to the parameter's data, and *valueSize* defines the size of that data.

RETURN CODES

- B_OK. No error.

- Other errors. An error occurred communicating with the Media Server, or with a node in the roster.

GetParameterValue() , SetParameterValue()

protected:

```
virtual status_t GetParameterValue(int32 parameterID, bigtime_t *lastChangeTime,
    void *value, size_t *ioSize) = 0
virtual void SetParameterValue(int32 parameterID, bigtime_t changeTime,
    const void *value, size_t size) = 0
```

You should implement `GetParameterValue()` to store the value of the parameter with the specified *parameterID* in the memory pointed to by *value*. The size_t value pointed to by *ioSize* specifies the size of the value buffer; prior to returning, your implementation of `GetParameterValue()` should change *ioSize* to the actual size of the returned data.

Also, you should set *lastChangeTime* to the time at which the control's value most recently changed.

`GetParameterValue()` should return **`B_OK`** when done, or an appropriate error code if something goes wrong.

Likewise, you should implement `SetParameterValue()` to change the value of the parameter; the *changeTime* argument is the performance time at which the change should occur; in other words, you may need to queue the request so it can be handled at the requested time. *value* points to the value to which the parameter should be set, and *size* is the number of bytes of data in the *value*.

It's possible that a single parameter may have several channels of values, if that parameter is a multi-channel parameter. For example, if the parameter is a two-channel slider (such as a stereo gain control, where the left and right channels are controlled individually within a single parameter), the *value* argument would point to an array of two floats, and size would be 8 (sizeof(float) * 2).

HandleMessage()

```
virtual status_t HandleMessage(int32 message, const void *data, size_t size)
```

When your node's service loop receives a *message*, in addition to passing it to BMediaNode and other superclasses of your node, you should also pass it to `BControllable::HandleMessage()`. You should start at the most-derived class' implementation of `HandleMessage()` and work your way upward until **`B_OK`** is returned.

If it's a message intended for the BControllable interface, it'll be dispatched and **`B_OK`** will be returned; otherwise, `HandleMessage()` will return an error so you know to try something else.

```
/* Message received */

if ((BControllable::HandleMessage(message, data, size) != B_OK) &&
        (BMediaNode::HandleMessage(message, data, size) != B_OK)) {
   BMediaNode::HandleBadMessage(message, data, size);
}
```

In this example, the BControllable implementation of `HandleMessage()` gets the first crack at handling the request. If it doesn't know what to do with the message, it's forwarded to BMediaNode's implementation. If the message still isn't handled, it's then sent to **`BMediaNode::HandleBadMessage()`** to be dealt with.

RETURN CODES

- B_OK. The message was dispatched.
- Other errors, as appropriate. Each *message* code may respond with various error codes.

See also: **`BMediaNode::HandleMessage()`**, "About Multiple Virtual Inheritance" on page21

LockParameterWeb() , UnlockParameterWeb()

bool **LockParameterWeb(**void**)**
void **UnlockParameterWeb(**void**)**

LockParameterWeb() locks the web to prevent access to it while you're working on it, and UnlockParameterWeb() releases it when you're done. You should surround your accesses to the web with these calls:

```
LockParameterWeb();
Web()->MakeGroup("EqualizerControls");
...
UnlockParameterWeb();
```

MakeParameterData() , ApplyParameterData()

protected:

status_t **MakeParameterData(**const int32 **parameterList*, int32 *numParameters*, void **buffer*, size_t **ioSize***)**
status_t **ApplyParameterData(**const void **value*, size_t *size***)**

The `MakeParameterData()` utility function takes a list of parameter IDs from *parameterList* and calls **`GetParameterValue()`** for each of them, storing the values in the specified *buffer* until the

size specified in *ioSize* is filled, or all the parameters are read. The number of bytes of the buffer used will be returned in *ioSize*.

If your BControllable is also a BBufferConsumer that accepts **B_MEDIA_PARAMETERS** type data on some input, call `ApplyParameterData()` with *value* set to the result of **BBuffer::Data()** and *size* set to **BBuffer::Size()**. This function will then parse the parameter change requests in the buffer and dispatch them to your **SetParameterValue()** function to fulfill the requests.

This lets your node support easy automation of parameter information. Even more benefit can be obtained by also deriving from BBufferProducer, and providing an output for the **B_MEDIA_PARAMETERS** data format, so that changes can be recorded as they occur. This provides a mechanism for automating the parameters by recording a user's changes to them, then playing back the changes later.

RETURN CODES

- B_OK. No errors.
- **B_NO_MEMORY**. The output *buffer* is too small.

SetParameterValue() see GetParameterValue()

SetParameterWeb() , Web()

protected:

status_t **SetParameterWeb(**BParameterWeb ***web)**
BParameterWeb ***Web(**void**)**

Your constructor should create a BParameterWeb object and call `SetParameterWeb()` with it as an argument. This will describe to the outside world what parameters are available and how they relate to each other; in other words, this describes your internal signal path, and how it can be manipulated.

If the web argument isn't **NULL**, and is different from the previously-established web for the BControllable node, a **B_MEDIA_WEB_CHANGED** message is sent to everyone watching for media notifications. See **StartWatching()** on for more information.

`SetParameterWeb()` will return **B_OK** if the web was set without errors; otherwise an error code will be returned.

The `Web()` function returns the BParameterWeb assigned to the BControllable.

StartControlPanel()

protected:

virtual status_t **StartControlPanel(**BMessenger **outMessenger***)**

This hook function is called whenever a client application wants the node to present its own control panel user interface (so that the user can configure the node).

On return, *outMessenger* is a BMessenger that you can use to communicate with teh control panel.

Because the add-on lives in the Media Server, and a problem in the user interface could bring down the entire system, it's recommended that the control panel run as its own team. This can be done easily by writing your node as both a Media Server add-on (by exporting `make_media_addon()`) and an application (by implementing `main()` and including **start_dyn.o** among the link libraries). Be sure you have the multi-launch application flags set on your add-on, or this won't work right.

Then your `StartControlPanel()` implementation can simply launch the add-on as an application, and if the user double-clicks the add-on, they'll be presented with the control panel. In addition, the user benefits by only having to install a single file for your add-on to work properly.

The first argv argument to your `main()` function will be a string of the format "node=%d" with the node ID in question as "%d".

The above implementation suggestion (providing your control panel by launching the add-on as an application) is the default behavior of `StartControlPanel()`, so if that's how you implement your BControllable, you don't have to override `StartControlPanel()` at all. In this case, the returned BMessenger is for the control panel application, and not for a particular BWindow or BView therein.

RETURN CODES

- B_OK. No errors occurred.
- `B_ERROR`. Node wasn't loaded from an add-on.
- `B_BAD_VALUE`. An error occurred locating the image from which the node was loaded, or the add-on can't be launched as an application.
- `B_LAUNCH_FAILED`. The control panel couldn't be launched for some other reason, such as insufficient memory.

- UnlockParameterWeb() see LockParameterWeb()
- Web() see SetParameterWeb()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BControllable Function Summary

Function Summary

Function	Description
ApplyParameterData()	Processes parameter change requests and calls SetParameterValue()
BroadcastChangedParameter()	Tell clients that a parameter's configuration has changed
BroadcastNewParameterValue()	Tells clients
BControllable()	Constructor
GetParameterValue()	Hook: return a parameter's value
HandleMessage()	Dispatches a message received on the control port
LockParameterWeb()	Locks the web
MakeParameterData()	Creates a buffer containing the values of all the node's parameters
SetParameterValue()	Sets a parameter's value
SetParameterWeb()	Sets the parameter web for the node
StartControlPanel()	Hook: present user interface to control the node
UnlockParameterWeb()	Unlocks the web

Web()	Returns the node's parameter web

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Game Kit

BFileGameSound

Declared in: be/game/FileGameSound.h
more...

BFileGameSound()

```
BFileGameSound(const char *pathname,
    bool looping = true,
    BGameSoundDevice *device = NULL)
```

This new constructor lets you set the file as a pathname (instead of as an **entry_ref**) .

SetPaused() , IsPaused()

```
virtual status_t SetPaused(bool pauseIt, bigtime_t rampTime)
int32 IsPaused(void)
```

SetPaused() lets you pause and resume the sound as *pauseIt* is **true** or **false**. *rampTime* is the amount of time, in microseconds, that will be used to fade in or fade out.

IsPaused() returns a value that describes the current paused state:

- **B_NOT_PAUSED**. The sound isn't paused.
- **B_PAUSED**. The sound is currently paused.
- **B_PAUSE_IN_PROGRESS**. The sound is fading out but isn't yet paused.

RETURN CODES

SetPaused() returns:

- B_OK. The sound was successfully paused or resumed.
- **EALREADY**. Can't pause a paused sound, or unpause one that's playing.
- **B_ERROR**. Resource error.

BSimpleGameSound

Declared in: be/game/SimpleGameSound.h

BSimpleGameSound()

```
BSimpleGameSound(const char *pathname,
    BGameSoundDevice *device = NULL)
```

This new constructor lets you set the file as a pathname (instead of as an `entry_ref`) .

The Be Book Release Notes Release Notes Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaEventLooper

Derived from: BMediaNode
Declared in: be/media/MediaEventLooper.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BMediaEventLooper class provides a control thread that automatically queues received media events and calls the `HandleEvent()` function you implement to process them as necessary. This takes a lot of the drudgery out of writing nodes:

- You don't have to write a service thread to receive and process incoming messages; BMediaEventLooper does this for you (much like a BLooper, hence the similarity in names).
- You don't have to write code to queue up events until it's time to handle them, since BMediaEventLooper maintains a BTimedEventQueue for you. Just push events onto the queue as you receive them and the BMediaEventLooper pops them off for you at the appropriate times and asks you to handle them.
- If all your node needs to do is know when `Start()`, `Stop()`, and `Seek()` requests come due, your node doesn't even have to override these functions—BMediaEventLooper automatically intercepts them and pushes them onto the queue for you.
- The BTimedEventQueue employed by BMediaEventLooper automatically provides support for queueing multiple start, stop, and seek events.

BMediaEventLooper maintains two queues: one for real-time events, and one for normal events. Events in the real-time queue are handled according to their real time, while events in the other queue are handled based on their performance time. So the values of the time stamps on events in these two queues are handled differently; keep this in mind if you need to peek into the queues yourself.

Implementing a Node (the Easy Way)

Although BMediaEventLooper does take away a lot of the complicated node construction work, there's still work to be done. There are still plenty of hooks you have to implement yourself (which ones, exactly, depend on whether you're a producer or a consumer).

You can study a specific example of how to implement a node using BMediaEventLooper by reading "A BMediaEventLooper Example".

Hook Functions

- `CleanUpEvent()`
- `ControlLoop()`
- `HandleEvent()`
- `OfflineTime()`

Constructor and Destructor

BMediaEventLooper()

protected:

```
explicit BMediaEventLooper(uint32 apiVersion = B_BEOS_VERSION)
```

You need to override this function to handle your node's initialization needs. The *apiVersion* argument indicates the version of the BeOS API the object was written for; you should let the default value be used—this will cause your node to inherit the API version of the BeOS version you're compiling under.

~BMediaEventLooper()

protected:

```
virtual ~BMediaEventLooper(void)
```

Calls **Quit()** to stop the control thread and free allocated memory.

Member Functions

BufferDuration() see SetBufferDuration()

CleanUpEvent()

protected:

```
virtual void CleanUpEvent(const media_timed_event *event)
```

Implement this function to clean up after custom events you've created and added to your queue. It's called when a custom event is removed from the queue, to let you handle any special tidying-up that the event might require.

ControlLoop()

protected:

```
virtual void ControlLoop(void)
```

This function waits for messages, pops events off the queue, and calls `DispatchEvent()`. It's called automatically when the BMediaEventLooper is `Run()`.

If you choose to reimplement this function, be very careful; it's very easy to cause Bad Things to happen.

ControlThread()

```
thread_id ControlThread(void)
```

Returns the control thread's thread_id.

DispatchEvent()

protected:

```
void DispatchEvent(const media_timed_event *event,
    bigtime_t lateness, bool realTimeEvent = false)
```

Calls `HandleEvent()` to let your code handle the specified *event*. If your code doesn't handle it, this function may have a default handler to process it. In general, you won't call this function.

BMediaEventLooper compensates your performance time by adding the event latency (see `SetEventLatency()`) and the scheduling latency (or, for real-time events, only the scheduling latency).

It's the control loop's job to remove the event from the queue; this function doesn't do that.

EventLatency() see SetEventLatency()

EventQueue() , RealTimeQueue()

protected:

```
BTimedEventQueue *EventQueue(void)
BTimedEventQueue *RealTimeQueue(void)
```

`EventQueue()` returns a pointer to the BTimedEventQueue used by the message handler. `RealTimeQueue()` returns a pointer to the BTimedEventQueue used to queue real-time events.

Events in the real-time queue are handled according to their real time, while events in the normal event queue are handled based on their performance time. So the values of the time stamps on events in these two queues are handled differently; keep this in mind if you need to peek into the queues yourself.

HandleEvent()

protected:

```
virtual void HandleEvent(const media_timed_event *event,
    bigtime_t lateness, bool realTimeEvent = false)
```

Implement this function to handle incoming media events. The *event* argument references a media_timed_event structure that describes the event. *lateness* indicates how late the event is, and *realTimeEvent* is true if the event needs to be handled in real time.

The BMediaEventLooper will call this function from the `DispatchEvent()` function. It's the control loop's job to remove the event from the queue.

OfflineTime() see SetOfflineTime()

NodeRegistered()

protected:

virtual void **NodeRegistered**(void)

The Media Server calls this hook function after the node has been registered. This is derived from BMediaNode; BMediaEventLooper implements it to call **Run()** automatically when the node is registered; if you implement `NodeRegistered()` you should call through to `BMediaEventLooper::NodeRegistered()` after you've done your custom operations.

Priority() see SetPriority()

Quit()

void **Quit**(void)

Closes the node's control port and closes the control thread. Blocks until the control thread is gone.

RealTimeQueue() see EventQueue()

Run()

void **Run**(void)

Spawns and runs the control thread; this is called automatically by the default **NodeRegistered()** implementation. If you override **NodeRegistered()**, be sure you call through to the default implementation, or call `Run()`.

RunState() see SetRunState()

SchedulingLatency()

protected:

bigtime_t **SchedulingLatency**(void) const

Returns the scheduling latnecy, in microseconds, of the node.

SetBufferDuration() , BufferDuration()

protected:

void **SetBufferDuration**(bigtime_t *duration*)

bigtime_t **BufferDuration**(void) const

`SetBufferDuration()` sets the duration of the node's buffers. The *duration* is clamped to 0 if it's less than 0.

`BufferDuration()` returns the duration of the nodes' buffers.

SetEventLatency() , EventLatency()

protected:

void **SetEventLatency**(bigtime_t *latency*)

bigtime_t **EventLatency**(void) const

`SetEventLatency()` sets the event latency. The event latency is the upstream and downstream algorithmic latency of your node—but should not include scheduling latency. This latency is taken into account by the BMediaEventLooper when deciding when to pop events off the queue for you to process.

`EventLatency()` returns the event latency.

SetOfflineTime() , OfflineTime()

protected:

void **SetOfflineTime**(bigtime_t *offlineTime*)

virtual bigtime_t **OfflineTime**(void)

`SetOfflineTime()` sets the time that `OfflineTime()` will return.

Augment `OfflineTime()` to compute the node's current time; it's called by the Media Kit when it's in offline mode. Update any appropriate internal information as well, then call through to the BMediaEventLooper implementation.

SetPriority() , Priority()

protected:

void **SetPriority**(int32 *priority*)

int32 **Priority**(void) const

`SetPriority()` sets the control thread's priority. Values less than 0 are clamped to 0, and values greater than 120 are clamped to 120.

`Priority()` returns the control thread's current priority.

SetRunState() , RunState()

protected:

void **SetRunState(**run_state *state***)**
run_state **RunState(**void**)** const

`SetRunState()` sets the node's current run state.

`RunState()` returns the current run state.

See also: `run_state`

Constants

run_state

Declared in: be/media/MediaEventLooper.h

Constant	Description
`B_IN_DISTRESS`	The node is experiencing problems.
`B_UNREGISTERED`	The node hasn't been registered with the Media roster yet.
`B_STOPPED`	The node isn't running.
`B_STARTED`	The node is running.
`B_QUITTING`	The node's in the process of shutting down.
`B_TERMINATED`	The node has been terminated.
`B_USER_RUN_STATES`	Base for user-defined run states.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaEventLooper Function Summary

Function Summary

Function	Description
BufferDuration()	Returns the duration of the node's buffers
CleanUpEvent()	Hook: implement to clean up after any custom events you've added to your queue
ControlLoop()	Hook: can be augmented to change the behavior of the control loop
ControlThread()	Returns the thread_id of the control thread
DispatchEvent()	Calls HandleEvent and does default event handling
EventLatency()	Returns the event latency
EventQueue()	Returns a pointer to the BTimedEventQueue used for events
HandleEvent()	Hook: implement to handle media events
BMediaEventLooper()	Constructor
~BMediaEventLooper()	Destructor
NodeRegistered()	Hook: called when the node is registered with the Media Kit
OfflineTime()	Hook: implement to determine the node's current time in offline mode
Priority()	Returns the control thread's priority
Quit()	Closes the control port and stops

	the control thread
RealTimeQueue()	Returns a pointer to the BTimedEventQueue used for real-time events
Run()	Spawns and runs the control thread
RunState()	Returns the node's run state
SchedulingLatency()	Returns the node's scheduling latency
SetBufferDuration()	Sets the duration of the node's buffers
SetEventLatency()	Sets the event latency
SetOfflineTime()	Sets the time to be returned by OfflineTime()
SetPriority()	Sets the control thread's priority
SetRunState()	Sets the node's run state

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimedEventQueue

Derived from: (none)
Declared in: be/media/TimedEventQueue.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BTimedEventQueue class provides an easy way to queue a sequence of events within your node. You can use it to queue up start, stop, and seek requests, and to queue up incoming data buffers in preparation for handling them. Each queue element is tagged with the time at which the event should be processed, and functions are provided for locating the next event that should be handled.

Although you shouldn't need to subclass BTimedEventQueue, there's no reason you can't do it.

Cleaning Up After Nodes

Each event has a cleanup flag associated with it that indicates what sort of special action needs to be performed when the event is removed from the queue. If this value is **B_NO_CLEANUP**, nothing is done. If it's **B_RECYCLE**, and the event is a **B_HANDLE_BUFFER** event, BTimedEventQueue will automatically recycle the buffer associated with the event.

If the cleanup flag is **B_DELETE** or is **B_USER_CLEANUP** or greater, the cleanup hook function will be called. You can implement and establish a cleanup hook function to handle deleting event data yourself. The hook function is of type **cleanup_hook**:

```
typedef void (*cleanup_hook)(void *context, bigtime_t time, int32 what,
         void *pointer, uint32 pointerCleanup, int64 data);
```

You specify the cleanup hook function by calling **SetCleanupHook()**, like this:

```
SetCleanupHook(MyCleanupFunction, contextPtr);
```

The *contextPtr* is a pointer that your cleanup hook function uses, and can contain whatever data you require.

Constructor and Destructor

BTimedEventQueue()

```
BTimedEventQueue()
```

Initializes a new, empty, timed event queue.

~BTimedEventQueue()

```
virtual ~BTimedEventQueue()
```

Flushes the queue and deallocates memory allocated by the object.

Member Functions

AddEvent()

```
status_t AddEvent(const media_timed_event &event)
```

Adds the specified *event* to the queue.

RETURN CODES

- B_OK. The event was added.
- **B_BAD_VALUE**. Unknown or invalid event type (you can't use **B_NO_EVENT** or **B_ANY_EVENT**).
- **B_NO_INIT**. The queue hasn't been initialized.

DoForEach()

```
status_t DoForEach(for_each_hook hook, void *context,
    bigtime_t time,
    time_direction direction,
    bool inclusive = true,
    int32 event = B_ANY_EVENT)
```

For each event in the queue matching the specified parameters, the *hook* function is called. The *context* pointer is passed through to the hook function, and may point to anything your hook function requires.

- Setting *direction* to **B_ALWAYS** indicates that all events of the type indicated by *event* should be processed. The *time* and includive arguments are ignored.
- Setting direction to **B_BEFORE_TIME** indicates that all matching events occurring before the specified *time* should be processed. If *inclusive* is **true**, events occurring at *time* are also processed.
- Setting direction to **B_AT_TIME** processes all matching events scheduled at the specified *time*. The *inclusive* argument is ignored.
- If direction is **B_AFTER_TIME**, all matching events scheduled to occur after the specified *time* are processed. If *inclusive* is **true**, events scheduled to occur at *time* are also processed.

This provides a means for you to scan through the queue and perform a particular act on every node (or all nodes of a certain type, or that occur at certain times). If you want to process all events, you can specify a time of **B_INFINITE_TIMEOUT** and a direction of **B_BEFORE_TIME**.

The hook function should return a queue_action value.

```
queue_action MyHook(media_timed_event *event, void *context) {
   if (event->data == 0) {
      /* countdown finished, do something */
      return BTimedEventQueue::B_REMOVE_EVENT;
   }
   event->data--;
   return BTimedEventQueue::B_NO_ACTION;
}
```

In this example, the hook function processes the event (and indicates on return that it should be removed from the queue) if the `data` field of the event structure is zero; otherwise `data` is decremented and the event is left alone.

RETURN CODES

- B_OK. The events were processed.
- **B_ERROR**. An undefined error occurred.

EventCount()

```
int32 EventCount(void) const
```

Returns the number of events in the queue.

FindFirstMatch()

```
const media_timed_event *FindFirstMatch(bigtime_t eventTime,
    time_direction direction,
    bool inclusive = true,
    int32 eventType = B_ANY_EVENT)
```

Searches the event queue for the first event matching the given specifications. The search begins at the time specified by *eventTime*, and progresses in the specified *direction*.

- Setting *direction* to **B_ALWAYS** indicates that all events of the type indicated by *event* should be scanned. The *eventTime* and includive arguments are ignored.
- Setting direction to **B_BEFORE_TIME** indicates that all matching events occurring before the specified *eventTime* should be scanned. If *inclusive* is **true**, events occurring at *eventTime* are also scanned.
- Setting direction to **B_AT_TIME** scans all matching events scheduled at the specified *eventTime*. The *inclusive* argument is ignored.
- If direction is **B_AFTER_TIME**, all matching events scheduled to occur after the specified *eventTime* are scanned. If *inclusive* is **true**, events scheduled to occur at *eventTime* are also

scanned.

If you want to scan all events, you can specify a time of **B_INFINITE_TIMEOUT** and a direction of **B_BEFORE_TIME**.

If no matching event is found, **NULL** is returned.

FirstEvent() , FirstEventTime()

```
const media_timed_event *FirstEvent(void) const
bigtime_t FirstEventTime(void) const
```

`FirstEvent()` returns the first event in the queue, without removing it from the queue.

`FirstEventTime()` returns the first event's time, in microseconds.

RETURN CODES

- B_OK. No error occurred.
- **B_NO_INIT**. The queue hasn't been initialized.
- **B_INFINITE_TIMEOUT**. The queue is empty (`FirstEventTime()` only).

FlushEvents()

```
status_t FlushEvents(bigtime_t time, time_direction direction,
    bool inclusive = true,
    int32 event = B_ANY_EVENT)
```

Flushes the specified events from the queue. You specify which events to flush by indicating a *time* from which events should be flushed, a *direction* in which to search for events to flush, and the type of events to flush:

- Setting *direction* to **B_ALWAYS** indicates that all events of the type indicated by *event* should be flushed. The *time* and includive arguments are ignored.
- Setting direction to **B_BEFORE_TIME** indicates that all matching events occurring before the specified *time* should be flushed. If *inclusive* is **true**, events occurring at *time* are also flushed.
- Setting direction to **B_AT_TIME** flushes all matching events scheduled at the specified *time*. The *inclusive* argument is ignored.
- If direction is **B_AFTER_TIME**, all matching events scheduled to occur after the specified *time* are flushed. If *inclusive* is **true**, events scheduled to occur at *time* are also flushed.

If you want to flush all events, you can specify a time of **B_INFINITE_TIMEOUT** and a direction of **B_BEFORE_TIME**.

RETURN CODES

- B_OK. The events were flushed.
- **B_ERROR**. An undefined error occurred.

HasEvents()

```
bool HasEvents(void)
```

Returns **true** if there are events in the queue, otherwise returns **false**.

LastEvent() , LastEventTime()

```
const media_timed_event *LastEvent(void) const
bigtime_t LastEventTime(void) const
```

`LastEvent()` returns the last event in the queue, without removing it from the queue.

`LastEventTime()` returns the last event's time, in microseconds.

RETURN CODES

- B_OK. No error occurred.
- **B_NO_INIT**. The queue hasn't been initialized.
- **B_INFINITE_TIMEOUT**. The queue is empty (`LastEventTime()` only).

RemoveEvent() , RemoveFirstEvent()

```
status_t RemoveEvent(const media_timed_event *event)
status_t RemoveFirstEvent(const media_timed_event *outEvent = NULL)
```

`RemoveEvent()` removes the specified *event* from the queue.

`RemoveFirstEvent()` removes the first event from the queue. If *outEvent* isn't **NULL**, the specified buffer is filled with a copy of the event that's been removed.

RETURN CODES

- B_OK. The event was removed.
- **B_NO_INIT**. The queue hasn't been initialized.

SetCleanupHook()

void **SetCleanupHook(**cleanup_hook *hook*, void **context***)*

Sets up the cleanup hook function specified by hook to be called for events as they're removed from the queue. The hook will be called only for events with **cleanup_flag** values of **B_DELETE** or **B_USER_CLEANUP** or greater.

Constants

cleanup_flag

Declared in: be/media/TimedEventQueue.h

Constant	Description
`B_NO_CLEANUP`	Don't clean up when the event is removed from the queue.
`B_RECYCLE_BUFFER`	BTimedEventQueue should recycle the buffer when the buffer event is removed from the queue.
`B_EXPIRE_TIMER`	Call `TimerExpired()` on the event's `data` field.
`B_USER_CLEANUP`	Base value for user-defined cleanup types.

These values define how BTimedEventQueue should handle removing events from the queue. If the flag is `B_USER_CLEANUP` or greater, the cleanup hook function is called when the event is removed.

event_type

Declared in: be/media/TimedEventQueue.h

Constant	Description
`B_NO_EVENT`	Don't push buffers of this type. It's a return value only.
`B_ANY_EVENT`	Don't push buffers of this type; it's used in searches only.
`B_START`	A start event.
`B_STOP`	A stop event.
`B_SEEK`	A seek event.
`B_WARP`	A warp event.
`B_TIMER`	A timer event.
`B_HANDLE_BUFFER`	Represents a buffer queued to be handled. The event's `pointer` is a pointer to a BBuffer object.
`B_DATA_STATUS`	Represents a data status event.
`B_HARDWARE`	A hardware event.
`B_PARAMETER`	The buffer contains changes to parameter values; pass it to `BControllable::ApplyParameterData()`.
`B_USER_EVENT`	Base value for user-defined events.

These values describe type types of events that a BTimedEventQueue can handle.

queue_action

Declared in: be/media/TimedEventQueue.h

Constant	Description
`B_DONE`	End the `DoForEach()` pass.
`B_NO_ACTION`	Do nothing for this event.
`B_REMOVE_EVENT`	Remove the event.
`B_RESORT_QUEUE`	Sort the queue again to ensure that events are still in the right order.

These queue action values are returned by the hook function used by `DoForEach()`; these values indicate what `DoForEach()` should do to the event after the hook has processed it.

---

time_direction

Declared in: be/media/TimedEventQueue.h

Constant	Description
`B_ALWAYS`	Matches events occurring at any time.
`B_BEFORE_TIME`	Matches events occurring before a specified time.
`B_AT_TIME`	Matches events occurring at a specified time.
`B_AFTER_TIME`	Matches events occurring after a specified time.

These values are used to indicate a search direction through the time continuum.

Defined Types

cleanup_hook

Declared in: be/media/TimedEventQueue.h

```
typedef queue_action (*cleanup_hook)(media_timed_event *event, void *context);
```

The cleanup_hook type is used to define a hook function called while removing an event from the queue; it's set by calling **SetCleanupHook()**.

for_each_hook

Declared in: be/media/TimedEventQueue.h

```
typedef queue_action (*for_each_hook)(media_timed_event *event, void *context);
```

The for_each_hook type is used to define a hook function called by **DoForEach()**.

media_timed_event

Declared in: be/media/TimedEventQueue.h

```
struct media_timed_event {
   media_timed_event();
   media_timed_event(bigtime_t inTime, int32 inType);
   media_timed_event(bigtime_t inTime, int32 inType, void *inPointer,
            uint32 inCleanup);
   media_timed_event(bigtime_t inTime, int32 inType,
            void *inPointer, uint32 inCleanup,
            int32 inData, int64 inBigdata,
            char *inUserData, size_t dataSize = 0);

   media_timed_event(const media_timed_event &clone);
   void operator=(const media_timed_event &clone);

   ~media_timed_event();

   bigtime_t event_time;
   int32 type;
   void *pointer;
   uint32 cleanup;
   int32 data;
```

```
    int64 bigdata;
    char user_data[64];
    uint32 _reserved_media_timed_event_[8];
};
```

Describes a media event:

- `event_time` indicates the time at which the event is scheduled to occur.
- `type` indicates the type of event, as an event_type.
- `pointer` is a pointer to event-specific data owned by the event.
- `cleanup` is the cleanup_flag value for the event.
- `data` is an event-specific 32-bit value.
- `bigdata` is an event-specific 64-bit value.
- `user_data` is user-defined data.

The Be Book The Media Kit The Media Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimedEventQueue Function Summary

Function Summary

Function	Description
AddEvent()	Adds an event to the queue
DoForEach()	Calls a specified hook function for each matching event in the queue
EventCount()	Returns the number of events in the queue
FindFirstMatch()	Finds the first event that matches given parameters
FirstEvent()	Returns the first event in the queue
FirstEventTime()	Returns the time of the first event in the queue
FlushEvents()	Flushes the specified events from the queue
HasEvents()	Returns true if there are events in the queue
LastEvent()	Returns the last event in the queue
LastEventTime()	Returns the time of the last event in the queue
RemoveEvent()	Removes an event from the queue
RemoveFirstEvent()	Removes the first event from the queue
SetCleanupHook()	Sets up a hook to handle cleaning up events as they're removed from the queue
BTimedEventQueue()	Constructor

~BTimedEventQueue()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

A BMediaEventLooper Example

Implementing a node using the BMediaEventLooper class to handle the scheduling issues makes your job much easier. So much so, in fact, that Be recommends that you not try to bypass BMediaEventLooper. This section provides an example of creating a node using BMediaEventLooper to handle the control port and event queueing.

When you implement a node using BMediaEventLooper, you have to derive from multiple classes: BMediaEventLooper, of course, and BControllable if you want your node to be user-configurable. Then you need to also derive from BBufferConsumer and/or BBufferProducer, depending on whether you create or receive buffers. And if your node can provide a time source, you would also need to derive from BTimeSource.

BMediaEventLooper doesn't eliminate your obligation to implement the pure virtual functions in all the classes you derive from, and you may still have to implement some or all of the other virtual functions as well. All BMediaEvent gives you is automatic queueing of received messages, and automatic management of your control port. This still eliminates a lot of repetitive and uninteresting work, and makes your job a lot easier.

Let's look at an example node. This node derives from BBufferConsumer, BControllable, and BMediaEventLooper, and logs media events to a disk file. The node's latency, among other parameters, is user-configurable, so you can use this node to simulate different loads on the system.

This section only shows key portions of the sample node. If you'd like to play with the full source code, you can download it from the Be web site at ftp://ftp.be.com/pub/samples/media_kit/LoggingConsumer.zip.

The Constructor and Destructor

Let's start at the very beginning (a very good place to start). The constructor has to perform the initialization for all the superclasses:

Constructor

```
LoggingConsumer::LoggingConsumer(const entry_ref &logFile)
   :   BMediaNode("LoggingConsumer"),
      BBufferConsumer(B_MEDIA_UNKNOWN_TYPE),
      BControllable(),
      BMediaEventLooper(),
      mLogRef(logFile), mWeb(NULL),
      mLateBuffers(0),
      mLatency(50 * 1000),      // default to 50 milliseconds
      mSpinPercentage(0.10),      // default to spinning 10% of total latency
      mPriority(B_REAL_TIME_PRIORITY),
      mLastLatencyChange(0),
      mLastSpinChange(0),
      mLastPrioChange(0) {
   mLogger = new LogWriter(logFile);
}
```

The constructors for BMediaNode, BBufferConsumer, BControllable, and BMediaEventLooper are all called through to here, to perform the standard initialization for each of these superclasses. In addition, an assortment of local variables are initialized.

Finally, the thread that handles actually writing into the log file is started; this is done by the LogWriter class' constructor. This class will be used to access the log file. We won't get into any specifics of how the LogWriter class works. Suffice it to say that the LogWriter has some public variables that affect the messages it logs to the file, and a `Log()` function that actually writes a new entry into the log file.

Destructor

```
LoggingConsumer::~LoggingConsumer() {
   BMediaEventLooper::Quit();
   SetParameterWeb(NULL);
   mWeb = NULL;
   delete mLogger;
}
```

The destructor, shown above, begins by stopping the BMediaEventLooper's control loop. This is done by calling **`BMediaEventLooper::Quit()`**. Next we delete the parameter web by calling **`BControllable::SetParameterWeb()`**; we set the parameter web to **`NULL`** instead of deleting it

ourselves because this lets the node handle cleanup—the node deletes the web for us in this case. We then set our local pointer to the web to `NULL` so we know it's been deleted.

Once all that's been done, we delete the logging thread. This is done last to avoid possible race conditions that could arise if an event arrives that would trigger a write to the log. By waiting until the BMediaEventLooper has been quit, we can avoid this potential problem.

BMediaNode Functions

We also need to implement some functions from the BMediaNode class. Other than **BMediaNode::AddOn()**, BMediaEventLooper provides default implementations for these functions. In many cases, you can just use the default implementations. We'll look at one custom implementation, so you can see how it's done.

AddOn()

```
BMediaAddOn* LoggingConsumer::AddOn(int32 *) const {
   return NULL;
}
```

The **BMediaNode::AddOn()** function's job is to return a pointer to the BMediaAddOn object that instantiated the node. In this case, we're contained within an application, so we return **NULL**. But if the node were created from a media node add-on, we'd return a pointer to that object here.

Other BMediaNode Functions

```
void LoggingConsumer::SetRunMode(run_mode mode) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   logMsg.runmode.mode = mode;
   mLogger->Log(LOG_SET_RUN_MODE, logMsg);

   BMediaEventLooper::SetRunMode(mode);
}
```

The **BMediaNode::SetRunMode()** function handles changing the run mode of the node. The BMediaEventLooper class handles this automatically, but you can augment the implementation if you need to.

In this case, as we do in all the other BMediaNode functions we implement, we simply log the call, then defer to BMediaEventLooper's implementation.

We call `TimeSource()->Now()` to get the current time, which is inserted into the log along with a description of the type of event that occurred.

The other BMediaNode functions we implement work similarly, logging the request, then deferring to the BMediaEventLooper implementation.

BControllable Functions

The BControllable functions we implement allow our node to be user-configurable; these functions are called when someone needs to know the current value of one of the parameters, or when the value of a parameter needs to be changed.

GetParameterValue()

```
status_t LoggingConsumer::GetParameterValue(int32 id,
         bigtime_t* last_change, void* value, size_t* ioSize) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   logMsg.param.id = id;
   mLogger->Log(LOG_GET_PARAM_VALUE, logMsg);

   if (*ioSize < sizeof(float)) return B_ERROR;

   switch (id) {
   case LATENCY_PARAM:
      *last_change = mLastLatencyChange;
      *((float*) value) = mLatency/1000;      // the BParameter reads milliseconds, not micros
      *ioSize = sizeof(float);
      break;

   case CPU_SPIN_PARAM:
      *last_change = mLastSpinChange;
      *((float*) value) = mSpinPercentage;
      *ioSize = sizeof(float);
      break;

   case PRIORITY_PARAM:
      *last_change = mLastPrioChange;
      *((int32*) value) = mPriority;
      *ioSize = sizeof(int32);
      break;

   default:
      return B_ERROR;
   }

   return B_OK;
}
```

The **BControllable::GetParameterValue()** function is called to obtain the current value for one of the node's configurable parameters. The parameter is specified by an ID number in the *id* argument. *value* points to a memory buffer in which the value should be stored, and *ioSize*

indicates the size of the buffer. Our job is to store the time at which the indicated parameter last changed into *last_change*, the new value into the buffer pointed at by *value*, and the actual size of the returned value into *ioSize*.

Our logging node begins by logging the request to the log file.

Then the real implementation begins by checking to be sure the space provided by the caller is big enough for the result. In this node, all values are four bytes, so this is easy to check, but in your node, you might have to check on a parameter-by-parameter basis.

Then, based on the *id*, the result is filled out, based on values we've cached from the last change to the parameter values (these would be set when the web receives the appropriate messages indicating that the values have changed). We'll look at the functioning of the BParameterWeb shortly.

SetParameterValue()

```
void LoggingConsumer::SetParameterValue(int32 id,
         bigtime_t performance_time, const void* value, size_t size) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   logMsg.param.id = id;
   mLogger->Log(LOG_SET_PARAM_VALUE, logMsg);

   // if it's one of our parameters, enqueue a "set parameter" event for handling at the appro
time
   switch (id) {
   case LATENCY_PARAM:
   case CPU_SPIN_PARAM:
   case PRIORITY_PARAM:
      {
         media_timed_event event(performance_time,
               BTimedEventQueue::B_PARAMETER, (void*) value,
               BTimedEventQueue::B_NO_CLEANUP, size, id, NULL);
         EventQueue()->AddEvent(event);
      }
      break;

   default:      // do nothing for other parameter IDs
      break;
   }
   return;
}
```

BControllable::SetParameterValue() is called when a request is made to change the value of a parameter. We begin, as usual, by logging the request.

The meat of this function is the switch statement, which enqueues a **`media_timed_event`** representing the request. We instantiate a new media_timed_event with the performance time at which the change is to take place, and all the other parameters needed when changing a parameter's value.

This is then inserted into the event queue by calling **`BTimedEventQueue::AddEvent()`** on the BMediaEventLooper's queue, which is returned by **`BMediaEventLooper::EventQueue()`**. When the time specified by *performance_time* arrives, this will be dequeued and sent along to **`BMediaEventLooper::HandleEvent()`** automatically.

The event is given the type **`BTimedEventQueue::B_PARAMETER`**. This type is used for parameter change events.

BBufferConsumer Functions

Next we come to the BBufferConsumer functions that we have to implement. These functions actually handle arriving buffers, and negotiations with producers.

HandleMessage()

```
status_t LoggingConsumer::HandleMessage(int32 message, const void *data,
         size_t size) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_HANDLE_MESSAGE, logMsg);

   return B_ERROR;
}
```

HandleMessage() should never be called if you're using a BMediaEventLooper (if it's called, a message was received that can't be handled at this inheritance level), so we log it and return an error.

AcceptFormat()

```
status_t LoggingConsumer::AcceptFormat(const media_destination &dest,
         media_format* format) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_ACCEPT_FORMAT, logMsg);

   if (dest != mInput.destination) return B_MEDIA_BAD_DESTINATION;

   return B_OK;
}
```

AcceptFormat() is called to confirm that a specified format is acceptable for a given destination. Your implementation needs to look at the format and make this decision, reporting an error if the format is unacceptable.

The example implementation logs the call, then checks to be sure the destination is in fact valid. In this example node, there's only one input allowed, so this is checked easily. If you have an array or linked list of inputs, you'll need to check them all. If the destination isn't recognized, return **B_MEDIA_BAD_DESTINATION**.

Return **B_OK** if the format is acceptable. In this case, all formats are acceptable.

GetNextInput()

```
status_t LoggingConsumer::GetNextInput(int32* cookie,
```

```
        media_input* out_input) {
   if (0 == *cookie) {
      mInput.format.type = B_MEDIA_UNKNOWN_TYPE;      // accept any format
      *out_input = mInput;
      *cookie = 1;
      return B_OK;
   }
   else return B_BAD_INDEX;
}
```

The **GetNextInput()** function is used to iterate through all the inputs your node provides. You should fill *out_input* with the a copy of the requested input's **media_input** structure. The *cookie* is used to specify which input is to be returned; the caller specifies a pointer to 0 the first time **GetNextInput()** is called, and you can set this value appropriately to keep track of where you are scanning your list of inputs.

In this case, there's only one input available, so we return **B_BAD_INDEX** if **cookie* is nonzero (which indicates that the end of the list has been reached). Otherwise the out_input is filled in, and the *cookie* is changed to 1 to indicate that the first input has been scanned. **B_OK** is returned to indicate success.

DisposeInputCookie()

```
void LoggingConsumer::DisposeInputCookie(int32 /*cookie*/ ) {
   /* handle disposing of your cookie here, if necessary */
}
```

If your cookie isn't a simple integer value, but is actually a pointer, you should dispose of it in your **DisposeInputCookie()** implementation. In this case, we do nothing, since the cookie is an integer.

BufferReceived()

```
void LoggingConsumer::BufferReceived(BBuffer* buffer) {
   bigtime_t bufferStart = buffer->Header()->start_time;
   bigtime_t now = TimeSource()->Now();
   bigtime_t how_early=bufferStart-EventLatency()-SchedulingLatency()-now;

   log_message logMsg;
   logMsg.now = now;
   logMsg.buffer_data.start_time = bufferStart;
   logMsg.buffer_data.offset = how_early;
   mLogger->Log(LOG_BUFFER_RECEIVED, logMsg);

   if (B_MEDIA_PARAMETERS == buffer->Header()->type) {
      ApplyParameterData(buffer->Data(), buffer->SizeUsed());
      buffer->Recycle();
   }
```

```
   else {
      status_t err;
      media_timed_event event(buffer->Header()->start_time,
               BTimedEventQueue::B_HANDLE_BUFFER,
               buffer, BTimedEventQueue::B_RECYCLE_BUFFER);
      err = EventQueue()->AddEvent(event);

      if (err) buffer->Recycle();
   }
}
```

`BufferReceived()` is called when buffers arrive on an active connection. Our primary task is to enqueue them for processing at the appropriate time. We begin by logging the arrival of the buffer, including information about the time at which it arrived and how early it arrived.

B_MEDIA_PARAMETERS type buffers have to be handled specially (each parameter change in the buffer has its own performance time listed), so we check the buffer's header to see if it's a **B_MEDIA_PARAMETERS** buffer. If it is, we call `ApplyParameterData()` to apply the parameter changes, and the buffer is immediately recycled so it can be reused.

If it's any other type of buffer, a new `media_timed_event` is created to reference the new buffer. The event is of type **BTimedEventQueue::B_HANDLE_BUFFER**, and the cleanup mode is **BTimedEventQueue::B_RECYCLE_BUFFER**, so that the buffer will automatically be recycled after the event is processed. Then the new event is enqueued.

If an error occurred trying to enqueue the buffer, it's not in the queue, so we need to recycle it ourselves by calling `BBuffer::Recycle()`. If we didn't do this, we'd leak buffers, which is an embarrassing problem for which there's no known cure. So take proper precautions.

ProducerDataStatus()

```
void LoggingConsumer::ProducerDataStatus(
         const media_destination & for_whom, int32 status,
         bigtime_t at_performance_time) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   logMsg.data_status.status = status;
   mLogger->Log(LOG_PRODUCER_DATA_STATUS, logMsg);

   if (for_whom == mInput.destination) {
      media_timed_event event(at_performance_time,
            BTimedEventQueue::B_DATA_STATUS, &mInput,
            BTimedEventQueue::B_NO_CLEANUP, status, 0, NULL);
      EventQueue()->AddEvent(event);
   }
}
```

ProducerDataStatus() is called when the upstream producer's status changes such that it will begin or stop sending buffers. This lets your node optimize its performance based on whether or not buffers are expected. After logging the call, the request is enqueued. Note that it's only queued if the specified destination, *for_whom*, matches one of the node's actual inputs. If it doesn't, the request should be ignored.

The event is given the type **BTimedEventQueue:B_DATA_STATUS**, and no cleanup is required.

GetLatencyFor()

```
status_t LoggingConsumer::GetLatencyFor(const media_destination &for_whom,
         bigtime_t* out_latency, media_node_id* out_timesource) {
   if (for_whom != mInput.destination) return B_MEDIA_BAD_DESTINATION;

   *out_latency = mLatency;
   *out_timesource = TimeSource()->ID();
   return B_OK;
}
```

GetLatencyFor()'s job is to report the latency for a particular destination. Make sure the destination is a valid one by checking all your inputs to see if any of them have that destination; if none match, return **B_MEDIA_BAD_DESTINATION**.

Otherwise, you should add up the algorithmic and downstream latency (but not the processing and scheduling latencies) for the input, and store that value in *out_latency*, and the node's time source in *out_timesource*, before returning **B_OK**. In our example node, the latency is `mLatency`, which is a user-configurable parameter.

The reason you don't need to include processing and scheduling latency in the result here is this: processing latency is compensated for by BMediaEventLooper beginning processing buffers one buffer's worth ahead of time up the chain. Scheduling latency is compensated for in each node.

In this case, the node has no downstream latency (it's not a producer), so the only latency that counts is our internal latency, which is `mLatency`. This is a user-configurable option in this node.

Connected()

```
status_t LoggingConsumer::Connected(
      const media_source& producer,
      const media_destination& where,
      const media_format& with_format,
      media_input* out_input) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_CONNECTED, logMsg);

   if (where != mInput.destination) return B_MEDIA_BAD_DESTINATION;
```

```
   // calculate my latency here, because it may depend on buffer sizes/durations, then
   // tell the BMediaEventLooper how early we need to get the buffers
   SetEventLatency(mLatency);

   // record useful information about the connection, and return success
   mInput.source = producer;
   *out_input = mInput;
   return B_OK;
}
```

BBufferConsumer::Connected() is called when a connection between the specified *producer*, and your node's destination *where* is established. The connection is logged to the log file.

If the destination isn't valid, **B_MEDIA_BAD_DESTINATION** is returned. Otherwise, the connection's latency is computed and **BMediaEventLooper::SetEventLatency()** is called to tell the BMediaEventLooper what the latency is. This is an important step, since the BMediaEventLooper class handles scheduling, it needs to know the latency!

In this case, we don't do much in the way of computing the latency; we have a known user-configurable value, `mLatency`, but in a real-world node, you need to compute this as accurately as you reasonably can.

Finally, important information about the connection is recorded into the input (the **media_source**, *producer*, for example), and we stuff a pointer to the input into *out_input*.

Disconnected()

```
void LoggingConsumer::Disconnected(
         const media_source &producer,
         const media_destination &where) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_DISCONNECTED, logMsg);

   ::memset(&mInput, 0, sizeof(mInput));
}
```

BBufferConsumer::Disconnected() is called when a connection between the source *producer* and the destination *where* is terminated. We log the call to the log file, then perform whatever tasks are necessary on disconnecting the nodes.

In this case, we zero out the **media_input** record used by the connection. Your node may have other tasks to perform, depending on your implementation.

FormatChanged()

```
status_t LoggingConsumer::FormatChanged(
```

```
        const media_source &producer,
        const media_destination &consumer,
        int32 change_tag,
        const media_format& format) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_FORMAT_CHANGED, logMsg);

   return B_OK;
}
```

BBufferConsumer::FormatChanged() is called whenever the media_format used for buffers transmitted on the connection between the given producer and consumer changes. This is called in response to your node calling **BBufferConsumer::RequestFormatChange()**; once the requested change has been made, this is called to let you know that it happened. *change_tag* matches the change tag specified returned by **RequestFormatChange()**, and the *format* indicates the negotiated new **media_format** for the connection.

Once your **FormatChanged()** function returns, future buffers will be in the new format, so you should prepare for the change at this point. In this case, we don't actually care about the contents of the received buffers, so we do nothing but log that the change occurred and return **B_OK**.

SeekTagRequested()

```
status_t LoggingConsumer::SeekTagRequested(
        const media_destination& destination,
        bigtime_t in_target_time,
        uint32 in_flags,
        media_seek_tag* out_seek_tag,
        bigtime_t* out_tagged_time,
        uint32* out_flags) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_SEEK_TAG, logMsg);

   return B_OK;
}
```

BBufferConsumer::SeekTagRequested() is called by the Media Server to obtain a seek tag corresponding to the specified *target_time*. Your node needs to cache these tags which are embedded in buffers it receives so it can return them when this function is called.

In this example, we don't support seek tags, so the request is logged and **B_OK** is returned. For more information, see **BBufferConsumer::SeekTagRequested()** and "Seek Tags".

BMediaEventLooper Functions

The node's core is the implementation of the various BMediaEventLooper virtual functions.

NodeRegistered()

```
void LoggingConsumer::NodeRegistered() {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_REGISTERED, logMsg);

   SetPriority(mPriority);
   Run();

   mInput.destination.port = ControlPort();
   mInput.destination.id = 0;
   mInput.node = Node();
   ::strcpy(mInput.name, "Logged input");

   mWeb = build_parameter_web();
   SetParameterWeb(mWeb);
}
```

Called when **BMediaRoster::RegisterNode()** is called, **BMediaEventLooper::NodeRegistered()**'s job is to set up the newly-registered node.

In this example, we begin by logging the call, then we set the thread's priority by calling **BMediaEventLooper::SetPriority()**, then we run the control thread by calling **BMediaEventLooper::Run()**. This thread will process events and pass them to our **BMediaEventLooper::HandleEvent()** function at the appropriate times.

Once that's done, we can initialize our inputs. In this case, we only have one, but your node might have multiple inputs. The input's **media_destination**'s port is set to **BMediaEventLooper::ControlPort()**. This tells the Media Kit which port to send messages to in order to communicate with the input. The destination's ID is set to a node-defined value—0 in this case, but your node should use a different value for each input you support. And the input's node is set to the result of **Node()**. We also set the input's name. This name should be unique for every input.

Finally, the node's parameter web is constructed by calling **build_parameter_web()**, and we call **BControllable::SetParameterWeb()** to establish the web.

Start(), Stop(), Seek(), TimeWarp()

These functions are called when the corresponding events occur. Usually you don't have to implement these, as your **HandleEvent()** implementation will receive an appropriate message

when the time comes to handle them. In fact, the only reason to implement these at all is if you want to know when one of these events is inserted into the queue.

If you do choose to implement them, you should call through to the base function. Let's look at the implementation of **`Start()`** as an example:

```
void LoggingConsumer::Start(bigtime_t performance_time) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_START, logMsg);

   BMediaEventLooper::Start(performance_time);
}
```

This code logs the request, then calls through to **`BMediaEventLooper::Start()`**. This will enqueue the request. We'll see how this is handled in **`HandleEvent()`** below.

HandleEvent()

This is the real meat of the node. **`HandleEvent()`** is called whenever the BMediaEventLooper detects that it's time for a queued event to be processed. You should implement the function to handle these events as they're dequeued. The specifics of your implementation will vary depending on what your node does.

```
void LoggingConsumer::HandleEvent(const media_timed_event *event,
         bigtime_t /* lateness */, bool /* realTimeEvent */) {
   log_message logMsg;
   logMsg.now = TimeSource()->Now();
   mLogger->Log(LOG_HANDLE_EVENT, logMsg);
```

First the call is logged, as is the routine for our example. Next, as we see below, comes a long switch statement that checks for and handles the various event types our node supports.

```
   switch (event->type) {
   case BTimedEventQueue::B_HANDLE_BUFFER:
      {
         BBuffer* buffer=const_cast<BBuffer*>((BBuffer*) event->pointer);
         if (buffer) {
            media_header* hdr = buffer->Header();
            if (hdr->destination == mInput.destination.id) {
               bigtime_t now = TimeSource()->Now();
               bigtime_t perf_time = hdr->start_time;

               bigtime_t how_early = perf_time - mLatency - now;

               logMsg.buffer_data.start_time = perf_time;
               logMsg.buffer_data.offset = how_early;
               logMsg.buffer_data.mode = RunMode();
```

```
            mLogger->Log(LOG_BUFFER_HANDLED, logMsg);

            if ((RunMode() != B_OFFLINE) &&            // lateness doesn't matter in offlin
               (RunMode() != B_RECORDING) &&      // ...or in recording mode
               (how_early < 0)) {
               mLateBuffers++;
               NotifyLateProducer(mInput.source, -how_early, perf_time);
            }
            else {
               // replace this with appropriate code for your node
               bigtime_t spin_start = ::system_time();
               bigtime_t spin_now = spin_start;
               bigtime_t usecToSpin = bigtime_t(mSpinPercentage / 100.0 * mLatency);
               while (spin_now - spin_start < usecToSpin) {
                  for (long k = 0; k < 1000000; k++) { /* intentionally blank */ }
                  spin_now = ::system_time();
               }
            }

            if ((B_OFFLINE == RunMode()) && (B_DATA_AVAILABLE == mProducerDataStatus)) {
               status_t err = RequestAdditionalBuffer(mInput.source, buffer);
               if (err) {
                  logMsg.error.error = err;
                  mLogger->Log(LOG_ERROR, logMsg);
               }
            }
         }
         else {
            /* wrong destination! */
         }

         buffer->Recycle();
      }
   }
   break;
```

The first case is the **`BTimedEventQueue::B_HANDLE_BUFFER`** event. This event is received when it's time to handle an incoming buffer.

The buffer is obtained by casting the event's pointer field into a BBuffer pointer. If it's **`NULL`**, nothing is done. Otherwise, the buffer's header is grabbed and stashed into `hdr`. If the destination is invalid, we do nothing (this should never happen, but better safe than sorry).

If the destination is good, it's time to begin processing the buffer. We compute how early the buffer arrived by subtracting the latency and the current performance time from the buffer's performance time.

Then the buffer's performance is logged (this step is of course specific to this particular example node).

If the buffer is late (`how_early` is negative) and we're in neither **B_OFFLINE** nor **B_RECORDING** mode, we ignore the buffer and notify the producer that the buffer was late by calling **BBufferConsumer::NotifyLateProducer()**. This lets the producer adjust its performance to attempt to avoid future buffers from arriving late. We allow tardiness in **B_OFFLINE** and **B_RECORDING** modes because lateness doesn't matter in these cases.

If the buffer is on time, we process the buffer. In this example, we just waste some processing time, rougly corresponding to our latency. In a real node, you'd replace this code with code that handles the buffer, whether by displaying it to the screen or playing it through speakers, or whatever might be appropriate.

If the node is in **B_OFFLINE** mode, and the cached producer data status is **B_DATA_AVAILABLE** (meaning the producer has more buffers to send us), we must call **BBufferConsumer::RequestAdditionalBuffer()** to tell the producer we're ready for another buffer. If we fail to call this, performance will suffer greatly. If **RequestAdditionalBuffer()** fails, we log the error.

Once processing of the buffer is done, we call **BBuffer::Recycle()** to make it available for future use.

The next possible event is the **BTimedEventQueue::B_PARAMETER**, which indicates a change to the parameters' values.

```
   case BTimedEventQueue::B_PARAMETER:
      {
         size_t dataSize = size_t(event->data);
         int32 param = int32(event->bigdata);
         logMsg.param.id = param;

         if (dataSize >= sizeof(float)) switch (param) {
         case LATENCY_PARAM:
            {
               float value = *((float*) event->user_data);
               mLatency = bigtime_t(value* 1000);
               mLastLatencyChange = logMsg.now;

               SetEventLatency(mLatency);

               SendLatencyChange(mInput.source, mInput.destination, EventLatency() +
SchedulingLatency());
               BroadcastNewParameterValue(logMsg.now, param, &value, sizeof(value));

               logMsg.param.value = value;
               mLogger->Log(LOG_SET_PARAM_HANDLED, logMsg);
            }
            break;
```

If the latency parameter is changed, we record the new latency in `mLatency`, and the time at which the change occurred in `mLastLatencyChange`, and we call **`BMediaEventLooper::SetEventLatency()`** to let the control thread know about the latency change.

In addition, the new latency is sent to the producer by calling **`BBufferConsumer::SendLatencyChange()`**. This lets the producer alter its behavior as appropriate to avoid running amok (or falling behind, depending on the new latency). And **`BControllable::BroadcastNewParameterValue()`** is called to share the change to the parameter's value with anyone that's watching for changes.

Finally, we log the change to disk.

```
        case CPU_SPIN_PARAM:
           {
              float value = *((float*) event->user_data);
              mSpinPercentage = value;
              mLastSpinChange = logMsg.now;
              BroadcastNewParameterValue(logMsg.now, param, &value, sizeof(value));
              logMsg.param.value = value;
              mLogger->Log(LOG_SET_PARAM_HANDLED, logMsg);
           }
           break;
```

Similarly, if the CPU spin parameter is changed, we record the new value and the time at which the change occurred, then broadcast the change to the world and log the change. Note that since this doesn't correspond to anything BMediaEventLooper cares about, we have no additional work to do.

```
        case PRIORITY_PARAM:
           {
              mPriority = *((int32*) event->user_data);
              SetPriority(mPriority);

              mLastPrioChange = logMsg.now;
              BroadcastNewParameterValue(logMsg.now, param, &mPriority, sizeof(mPriority));
              logMsg.param.value = (float) mPriority;
              mLogger->Log(LOG_SET_PARAM_HANDLED, logMsg);
           }
           break;
```

When the priority parameter is changed, we call **`BMediaEventLooper::SetPriority()`** to tell the control thread about its new priority. You must never directly change the thread's priority, since the priority affects the functioning of the node, and there are other changes that have to be made based upon the change.

The new value is saved, along with the change time, and the value is broadcast and the log entry is recorded to disk.

```
        default:
           mLogger->Log(LOG_INVALID_PARAM_HANDLED, logMsg);
           break;
        }
     }
     break;
```

Here we deal with the case where a nonexistent parameter was changed. This error condition is logged to disk. You should deal gracefully with this possibility, although it should never happen.

The next event type that needs to be handled is **`BTimedEventQueue::B_START`**, which is received when it's time to start processing buffers:

```
case BTimedEventQueue::B_START:
   mLogger->Log(LOG_START_HANDLED, logMsg);
   break;
```

In this example node, we simply log the start request. Your node should insert whatever code is necessary to prepare to process buffers. If your node is a producer, you should start sending buffers at this point. Be sure to call **`BBufferProducer::SendDataStatus()`** to let the consumer know that you're sending buffers.

The next event type is **`BTimedEventQueue::B_STOP`**, which is received when it's time to stop processing buffers:

```
case BTimedEventQueue::B_STOP:
   mLogger->Log(LOG_STOP_HANDLED, logMsg);
   EventQueue()->FlushEvents(0, BTimedEventQueue::B_ALWAYS, true,
         BTimedEventQueue::B_HANDLE_BUFFER);
   break;
```

Stopping your node implies that any buffers you've received to date that haven't been processed yet should be ignored, so the event queue is flushed of all **`BTimedEventQueue::B_HANDLE_BUFFER`** events. If you have other tasks that need to be performend when your node is stopped, this is the place to handle them.

Note that if your node is a producer, you should call **`BBufferProducer::SendDataStatus()`** here to let the consumer know that no more buffers will be arriving.

The **`BTimedEventQueue::B_SEEK`** request is received when it's time to perform a seek operation on the node's media.

```
case BTimedEventQueue::B_SEEK:
   mLogger->Log(LOG_SEEK_HANDLED, logMsg);
   break;
```

Handle the seek operation here. In this node, we just log the seek request.

The **BTimedEventQueue::B_WARP** request is received when it's time to perform a time warp operation on the node'.

```
   case BTimedEventQueue::B_WARP:
      mLogger->Log(LOG_WARP_HANDLED, logMsg);
      break;
```

In this example, we just log the request.

The **BTimedEventQueue::B_DATA_STATUS** event is received when the producer's **BBufferProducer::SendDataStatus()** function is called. Obviously you only need to handle this event if you're a consumer:

```
   case BTimedEventQueue::B_DATA_STATUS:
      mProducerDataStatus = event->data;
      logMsg.data_status.status = event->data;
      mLogger->Log(LOG_DATA_STATUS_HANDLED, logMsg);
      break;
```

In our node, we record the producer's current data status in the member variable `mProducerDataStatus`. We use this information to determine whether or not we're expecting buffers. Your node might wish to alter its priority or perform other optimizations if the producer stops sending buffers, or increase priority when buffers start arriving again. We also log the change.

```
   default:
      logMsg.unknown.what = event->type;
      mLogger->Log(LOG_HANDLE_UNKNOWN, logMsg);
      break;
   }
}
```

Finally, we log any messages received that we don't understand. In general you can ignore messages your node isn't prepared to handle.

Creating a Parameter Web

A parameter web describes to the Media Kit the various user-configurable options that might interest them, and how they should appear in a user interface. This information is used by the media theme to render the interface that the user sees when configuring the node via the preference panel instantiated by calling **BMediaRoster::StartControlPanel()**.

Creating a BParameterWeb is simple. You begin by creating the BParameterGroups that logically group related parameters, then you insert parameters into each group by calling the appropriate BParameterGroup functions for the various types of parameters that can be created.

BParameterGroup::MakeNullParameter() can be used to create a label or other non-configurable control that might appear in the interface.

BParameterGroup::MakeContinuousParameter() creates a parameter with a floating-point value, which is usually set using a slider control.

BParameterGroup::MakeDiscreteParameter() creates a parameter with a set of discrete possible values. These are usually displayed as pop-up menus, although some themes might use radio buttons or lists.

```
static BParameterWeb* build_parameter_web() {
   BParameterWeb* web = new BParameterWeb;

   BParameterGroup* mainGroup = web->MakeGroup("LoggingConsumer Parameters");
   BParameterGroup* group = mainGroup->MakeGroup("Latency Control");
   BParameter* nullParam = group->MakeNullParameter(INPUT_NULL_PARAM, B_MEDIA_NO_TYPE, "Latenc
B_GENERIC);
   BParameter* latencyParam = group->MakeContinuousParameter(LATENCY_PARAM, B_MEDIA_NO_TYPE, "
      B_GAIN, "msec", 5, 100, 5);
   nullParam->AddOutput(latencyParam);
   latencyParam->AddInput(nullParam);

   group = mainGroup->MakeGroup("CPU Percentage");
   nullParam = group->MakeNullParameter(CPU_NULL_PARAM, B_MEDIA_NO_TYPE, "CPU Spin Percentage"
B_GENERIC);
   BContinuousParameter* cpuParam = group->MakeContinuousParameter(CPU_SPIN_PARAM, B_MEDIA_NO_
      B_GAIN, "percent", 5, 80, 5);
   nullParam->AddOutput(cpuParam);
   cpuParam->AddInput(nullParam);

   group = mainGroup->MakeGroup("Priority");
   nullParam = group->MakeNullParameter(PRIO_NULL_PARAM, B_MEDIA_NO_TYPE, "Thread Priority",
B_GENERIC);
   BDiscreteParameter* prioParam = group->MakeDiscreteParameter(PRIORITY_PARAM, B_MEDIA_NO_TYP
B_GENERIC);
```

```
    prioParam->AddItem(5, "B_LOW_PRIORITY");
    prioParam->AddItem(10, "B_NORMAL_PRIORITY");
    prioParam->AddItem(15, "B_DISPLAY_PRIORITY");
    prioParam->AddItem(20, "B_URGENT_DISPLAY_PRIORITY");
    prioParam->AddItem(100, "B_REAL_TIME_DISPLAY_PRIORITY");
    prioParam->AddItem(110, "B_URGENT_PRIORITY");
    prioParam->AddItem(120, "B_REAL_TIME_PRIORITY");

    return web;
}
```

Notice the use of the **BDiscreteParameter::AddItem()** function to add the discrete values to the priority parameter. Each item has a value and a label that's displayed in the user interface. In this example, these correspond to the various thread priorities and their names.

The user interface provided by this node lets the user configure the behavior of the node, including the thread priority, how busy the CPU should be while a buffer is "processed," and the latency of buffer processing.

Obviously your node will have different parameters (and may have none, in which case you wouldn't even derive from BControllable, and wouldn't need a BParameterWeb at all).

This function is called from the **BMediaEventLooper::NodeRegistered()** implementation above. `NodeRegistered()` then calls **SetParameterWeb()** to establish the web for use by `StartControlPanel()` and other functions that use the web.

This node's web, as interpreted by the default system theme in Release 4.5, looks like this:

Producer-specific Issues

There are obviously some additional virtual functions you need to implement if your node derives from BBufferProducer.

Connect()

Your **BBufferProducer::Connect()** implementation should call **BMediaEventLooper::SetEventLatency()** to establish your total latency. This value, which is your internal latency plus the downstream latency, is used by BMediaEventLooper to determine when to pop events off the queue for you to process. By keeping this value up-to-date, you can improve performance. This code might resemble the following:

```
/* calculate processing latency */

bigtime_t latency = calculate_buffer_latency();
latency += estimate_max_scheduling_latency();

/* calculate downstream latency */

bigtime_t downstream;
media_node_id timesource;
FindLatencyFor(output.destination, &downstream, &timesource);

bigtime_t totalLatency = latency + downstream;
SetEventLatency(totalLatency);
```

The Rest of the Story...

The LogWriter class' implementation isn't critical to this example, and is cleverly ignored in the name of saving space. If you're curious, you can reference the complete sample code for the LoggingApplication, at ftp://ftp.be.com/pub/samples/media_kit/LoggingConsumer.zip.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameter

Derived from: BFlattenable
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BParameter class is the superclass from which all controls representing configurable parameters within a BControllable node are derived.

The word "parameter," in terms of the Media Kit, refers to a point at which the data manipulation performed by a BControllable node can be configured, and doesn't actually refer to a user interface object. BParameter-derived objects don't implement a user interface. See the BMediaTheme class for information on how to create a user interface for a BControllable node.

The BParameter class is an abstract class; you'll normally use the `MakeNullParameter()`, `MakeDiscreteParameter()`, and `MakeContinuousParameter()` functions in the BParameterGroup class to actually create parameters and add them toa BParameterWeb. For more detailed information on parameters and how to create a BParameterWeb, see the BParameterGroup and BParameterWeb classes.

Hook Function

- `ValueType()`

Constructor and Destructor

BParameter objects may only be created by using the appropriate functions in the BParameterGroup class.

Member Functions

AddInput() , AddOutput()

status_t **AddInput(**BParameter **input***)**
status_t **AddOutput(**BParameter **output***)**

`AddInput()` adds the BParameter specified by *input* to the set of BParameters that transmit data into the BParameter. The specified *input* parameter's `AddOutput()` function is called to let it know that it's sending data to this parameter.

`AddOutput()` adds the BParameter specified by *output* to the set of BParameters to which the BParameter outputs data. The specified *output* parameter's `AddInput()` function is called to let it know that it's receiving data from this parameter.

RETURN CODES

- B_OK. The item was successfully added to the list.
- **B_ERROR**. The item couldn't be added to the list.

CountChannels() , SetChannelCount()

int32 **CountChannels(**void**)**
void **SetChannelCount(**int32 *numChannels***)**

Some BParameters have more than one channel of the same type. For example, a stereo volume parameter with independent left and right gain would have two channels, while a single parameter value that affects both the left and right by the same amount would have only one channel. `CountChannels()` returns the number of channels manipulated by the BParameter.

The default is one channel; if your node provides more channels, call `SetChannelCount()` after creating the BParameter. `SetChannelCount()` specifies the number of channels the parameter manipulates.

CountInputs() , CountOutputs()

int32 **CountInputs(**void**)**
int32 **CountOutputs(**void**)**

These two functions return the number of inputs or outputs currently connected to the BParameter. The number of inputs or outputs may be different from the number of channels,

since an input or output might be comprised of multiple channels (a single stereo sound input or output might consist of a left channel and a right channel, for example).

GetValue() , SetValue()

```
status_t GetValue(void *buffer, size_t *ioSize, bigtime_t *lastChange)
status_t SetValue(const void *buffer, size_t size, bigtime_t changeWhen)
```

`GetValue()` returns the current value of the BParameter. On entry, *buffer* points to *ioSize* bytes of memory, indicating where the current value of the BParameter should be stored. On return, *ioSize* indicates the actual number of bytes returned in the buffer, and *lastChange* indicates the time at which the BParameter's value last changed.

`SetValue()` changes the value of the BParameter. The new value, which is *size* bytes long, is obtained from the specified *buffer*. The change occurs at the performance time specified by *changeWhen*.

In either case, if the parameter is a multichannel parameter, the *buffer* should be large enough to contain the values for all the channels, and *ioSize* or *size* should indicate the total size of the buffer.

These calls simply dispatch to the **`BControllable::GetControlValue()`** and **`BControllable::SetControlValue()`** functions.

RETURN CODES

- B_OK. No errors.
- **`B_NO_MEMORY`**. The *buffer* is too small (`GetValue()`).
- **`B_BAD_VALUE`**. The parameter doesn't have a value (`GetValue()`).
- Port errors. <<<insert reference to Kernel Kit here>>>

Group()

```
BParameterGroup *Group(void) const
```

Returns the BParameterGroup that most directly contains the BParameter.

ID()

```
int32 ID(void) const
```

Returns the BParameter's ID number. This ID is used by the node to route value change requests to the right place. This value should be unique within the BParameterWeb

containing the BParameter.

InputAt() , OutputAt()

```
BParameter *InputAt(int32 index)
BParameter *OutputAt(int32 index)
```

`InputAt()` returns the BParameter that feeds data into the input number specified by *index*, which can range from zero to **`CountInputs()`**-1.

`OutputAt()` returns the BParameter that receives data from the output number specified by *index*, which can range from zero to **`CountOutputs()`**-1.

Both functions return **`NULL`** if an invalid *index* is specified.

Kind()

```
const char *Kind(void) const
```

Returns the BParameter's kind. The kind isn't necessarily user-readable, but it can be used to figure out what kind of BControl to create to visually represent the configuration option provided by the BParameter.

See the **`BParameterGroup::MakeNullParameter()`**, **`BParameterGroup::MakeDiscreteParameter()`**, and **`BParameterGroup::MakeContinuousParameter()`** functions for more detailed information about the available kinds.

MediaType() , SetMediaType()

```
media_type MediaType(void)
void SetMediaType(media_type type)
```

`MediaType()` returns the type of media data that flows through the BParameter.

`SetMediaType()` changes the recorded type of media data. The default, if it hasn't been changed, is **`B_MEDIA_NO_TYPE`**.

The media_type of a BParameter is used to inform interested parties as to the format of the data; it's an informational setting, and doesn't alter the data flowing through the BParameter in any way.

Name()

```
const char *Name(void) const
```

Returns the BParameter's name, which is suitable for display to the user (for example, it can be used as the label in a user interface object).

- OutputAt() see InputAt()
- Channels()
- Set see MediaType()
- SetValue() see GetValue()

- Type()

```
media_parameter_type Type(void) const
```

Returns the kind of parameter the object represents.

Constant	Description
B_NULL_PARAMETER	The BParameter is a BNullParameter.
B_DISCRETE_PARAMETER	The BParameter is a BDiscreteParameter
B_CONTINUOUS_PARAMETER	The BParameter is a BContinuousParameter

---

- Unit()

```
const char *Unit(void) const
```

Returns the unit of measurement used by the BParameter. This might be, for example, "dB," "kHz," or "fps." It should be human-readable, as it may be displayed in a user interface.

- ValueType()

```
virtual type_code ValueType(void) = 0
```

Returns a type code indicating the type of data type of the BParameter's value. This is usually B_INT32_TYPE for selectors or B_FLOAT_TYPE for sliders.

- Web()

```
BParameterWeb *Web(void) const
```

Returns the BParameterWeb that contains the BParameter.

Constants

Parameter Kinds

Declared in: be/media/ParameterWeb.h

Constant	Description
B_MASTER_GAIN	The parameter represents the main volume control.
B_GAIN	The parameter represents a gain control.
B_BALANCE	The parameter represents a balance control.
B_FREQUENCY	The parameter represents a frequency, like a radio tuner.
B_LEVEL	The parameter represents a level, as in EQ and effects.
B_SHUTTLE_SPEED	The parameter represents playback speed. A value of 1.0 indicates normal speed; less than 1.0 is slower, greater than 1.0 is faster.
B_CROSSFADE	The parameter indicates a crossfade for mixing audio or video; 0 indicates that the the first of a pair of streams should be presented, 100 indicates that the other should be presented, and values in between indicate the degree of mixing that should occur.
B_COMPRESSION	The parameter indicates compression ratio; the value is 0 for no compression, 99 for 100:1 compression.
B_QUALITY	The parameter indicates quality level; the value is 0 for maximum compression, 100 for no compression.
B_BITRATE	The parameter indicates bit rate in bits per second.
B_GOP_SIZE	The parameter indicates a "group of pictures" such as how many frames apart a keyframe should be inserted into a video stream.
B_MUTE	The parameter represents a mute control; a value of 0 passes data through unchanged, while a value of 1 mutes the data.
B_ENABLE	The parameter represents an enable toggle; a value of 0 disables the function while a value of 1 enables it.
B_INPUT_MUX	The parameter represents an input MUX. The value specifies which input should pass through.
B_OUTPUT_MUX	The parameter represents an output MUX; the value specifies which output should receive the incoming data.
B_TUNER_CHANNEL	The parameter represents a channel tuner (like a TV channel); the value indicates the channel number.
B_TRACK	The parameter's value indicates a track number.
B_RECSTATE	The parameter indicates whether the node is silent (0), playing (1), or recording (2).
B_SHUTTLE_MODE	The parameter indicates performance mode: -1 for backwards, 0 for stop, 1 for playing, and 2 for paused.
B_RESOLUTION	The parameter indicates video or audio resolution.

B_COLOR_SPACE	The parameter indicates video color space.
B_FRAME_RATE	The parameter represents a selector for picking frame rate.
B_VIDEO_FORMAT	The parameter represents a selector for choosing a video format.
B_WEB_PHYSICAL_INPUT	The parameter represents a physical input (such as a microphone jack or line input jack).
B_WEB_PHYSICAL_OUTPUT	The parameter represents a physical output (such as a line output jack or headphone jack).
B_WEB_LOGICAL_INPUT	The parameter represents a point at which bits of data are transferred between the computer and the A/V input hardware.
B_WEB_LOGICAL_OUTPUT	The parameter represents a point at which bits of data are transferred between the computer and the A/V output hardware.
B_WEB_ADC_CONVERTER	The parameter represents an analog to digital converter.
B_WEB_DAC_CONVERTER	The parameter represents a digital to analog converter.
B_WEB_BUFFER_INPUT	The parameter represents a media_input.
B_WEB_BUFFER_OUTPUT	The parameter represents a media_output.
B_GENERIC	The parameter's kind isn't one of the above.
B_SIMPLE_TRANSPORT	A discrete parameter indicating one of five states: rewinding, stopped, playing, paused, and fast-forwarding.

Indicates the kind of parameter represented by a BParameter object.

The **B_VIDEO_FORMAT** kind has specific values associated with it:

Value	Video Format
1	NTSC-M
2	NTSC-J
3	PAL-BDGHI
4	PAL-M
5	PAL-N
6	SECAM
7	MPEG-1
8	MPEG-2

---

media_parameter_type

Declared in: be/media/ParameterWeb.h

Constant	Description
B_NULL_PARAMETER	The BParameter is a BNullParameter.
B_DISCRETE_PARAMETER	The BParameter is a BDiscreteParameter
B_CONTINUOUS_PARAMETER	The BParameter is a BContinuousParameter

These are the possible parameter types, which are returned by the `Type()` function. They indicate what subclass of BParameter the object is.

Keep in mind that because these constants are members of the BParameter class, if you need to reference them from outside a BParameter, you need to preface the reference with "BParameter::", such as `BParameter::B_NULL_PARAMETER`.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameter Function Summary

Function Summary

Function	Description
AddInput()	Adds a BParameter to the set of inputs to the BParameter
AddOutput()	Adds a BParameter to the set of outputs from the BParameter
CountChannels()	Returns the number of channels the control handles
CountInputs()	Returns the number of inputs currently connected to the BParameter
CountOutputs()	Returns the number of outputs currently connected to the BParameter
GetValue()	Returns the BParameter's current value
Group()	Returns the BParameterGroup that owns the BParameter
ID()	Returns the BParameter's ID number
InputAt()	Returns the input at the specified index
Kind()	Returns the kind of the BParameter
MediaType()	Returns the type of media data the control handles
Name()	Returns the BParameter's name
OutputAt()	Returns the output at the specified index
SetChannelCount()	Sets the number of channels the control handles
SetMediaType()	Sets the type of media data the control

	handles
SetValue()	Sets the BParameter's value
Type()	Returns the type of control the BParameter represents
Unit()	Returns the unit of measurement used by the BParameter
ValueType()	Returns a type code representing the type of data kept by this control
Web()	Returns the BParameterWeb that owns the BParameter

The Be Book **The Media Kit** **The Media Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameterGroup

Derived from: BFlattenable
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

A BParameterGroup object serves as a logical grouping of BParameters. Each BParameter belongs to exactly one BParameterGroup, and is in fact created by the BParameterGroup object to which it belongs, via the `BParameterWeb::MakeGroup()` function.

BParamaeterGroups can be nested one inside another.

A BParameterGroup object that's owned by a BParameterWeb is a logical candidate for a pane in a multi-pane control panel such as a BTabView, wherein each tab causes another BParameterGroup's BParameters to be displayed.

Constructor and Destructor

BParameterGroup()

```
BParameterGroup()
```

Rather than directly creating a BParameterGroup object via the new operator, you should call `BParameterWeb::MakeGroup()` to create a top-level group, or **`BParameterGroup::MakeGroup()`** if you want to create a subgroup.

~BParameterGroup()

```
~BParameterGroup()
```

You don't need to delete a BParameterGroup object; the BParameterWeb that owns it will do this for you when the web is deleted.

Member Functions

CountGroups()

```
int32 CountGroups(void)
```

Returns the number of subgroups in the group. This count doesn't subgroups nested within subgroups.

CountParameters()

```
int32 CountParameters(void)
```

Returns the number of BParameters in the group. This count doesn't include BParameters in subgroups.

GroupAt()

```
BParameterGroup *ParameterGroup(int32 index)
```

Returns the subgroup at the specified *index* within the BParameterGroup. If the *index* is negative, or is greater than `CountGroups()`-1, `NULL` is returned.

MakeNullParameter()

```
BNullParameter *MakeNullParameter(int32 id, media_type type,
    const char *name, const char *kind)
```

Creates a new BNullParameter within the group, with the internal ID specified by *id*, which should be unique within the BParameterWeb that owns the group.

The type of media data that travels through the null parameter is indicated by *type*, which could be `B_MEDIA_UNKNOWN_TYPE` or `B_MEDIA_NO_TYPE`. The parameter's *name* will be displayed by client applications that display information about the node, and the parameter has the specified *kind*.

Constant	Description
`B_WEB_PHYSICAL_INPUT`	The parameter represents a physical input (such as a microphone jack or line input jack).
`B_WEB_PHYSICAL_OUTPUT`	The parameter represents a physical output (such as a line output jack or headphone jack).
`B_WEB_LOGICAL_INPUT`	The parameter represents a point at which bits of data are

	transferred between the computer and the A/V input hardware.
B_WEB_LOGICAL_OUTPUT	The parameter represents a point at which bits of data are transferred between the computer and the A/V output hardware.
B_WEB_ADC_CONVERTER	The parameter represents an analog to digital converter.
B_WEB_DAC_CONVERTER	The parameter represents a digital to analog converter.
B_WEB_BUFFER_INPUT	The parameter represents a media_input.
B_WEB_BUFFER_OUTPUT	The parameter represents a media_output.
B_GENERIC	The parameter's kind isn't one of the above.

These kinds are used when a control panel needs to draw a signal flow diagram, so it can use the appropriate symbols for these points in the flow.

If you create your own kind, you won't break anything, but you should contact Be Developer Support for guidance, so we can standardize given your needs.

See also: the BNullParameter class.

MakeDiscreteParameter()

BContinuousParameter ***MakeContinuousParameter(**int32 *id*, media_type *type*,
const char **name*, const char **kind***)**

Creates a new BDiscreteParameter object and attaches it to the group. The BDiscreteParameter will have the specified internal *id*, which should be unique within the BParameterWeb that owns the group.

The BDiscreteParameter will affect media data of the specified *type*, and will have the specified *name*.

The kind of discrete parameter is specified by *kind*, and may be any of the following values:

Constant	Description
B_MUTE	The parameter represents a mute control; a value of 0 passes data through unchanged, while a value of 1 mutes the data.
B_ENABLE	The parameter represents an enable toggle; a value of 0 disables the function while a value of 1 enables it.
B_INPUT_MUX	The parameter represents an input MUX. The value specifies which input should pass through.
B_OUTPUT_MUX	The parameter represents an output MUX; the value specifies which output should receive the incoming data.
B_TUNER_CHANNEL	The parameter represents a channel tuner (like a TV channel); the value indicates the channel number.
B_TRACK	The parameter's value indicates a track number.
B_RECSTATE	The parameter indicates whether the node is silent (0), playing (1), or

	recording (2).
B_SHUTTLE_MODE	The parameter indicates performance mode: -1 for backwards, 0 for stop, 1 for playing, and 2 for paused.
B_RESOLUTION	The parameter indicates video or audio resolution.
B_COLOR_SPACE	The parameter indicates video color space.
B_FRAME_RATE	The parameter represents a selector for picking frame rate.

MakeContinuousParameter()

BContinuousParameter ***MakeContinuousParameter(**int32 *id*, media_type *type*,
const char **name*, const char **kind*, const char **unit*,
float *minValue*, float *maxValue*, float *step***)**

Creates a new BContinuousParameter object and attaches it to the group. The BContinuousParameter will have the specified internal *id*, which should be unique within the BParameterWeb that owns the group.

The BContinuousParameter will affect media data of the specified *type*, and will have the specified *name*.

The kind of parameter the BContinuousParameter represents is specified by *kind*, and may be any of the following values:

Constant	Description
B_MASTER_GAIN	The parameter represents the main volume control.
B_GAIN	The parameter represents a gain control.
B_BALANCE	The parameter represents a balance control.
B_FREQUENCY	The parameter represents a frequency, like a radio tuner.
B_LEVEL	The parameter represents a level, as in EQ and effects.
B_SHUTTLE_SPEED	The parameter represents playback speed. A value of 1.0 indicates normal speed; less than 1.0 is slower, greater than 1.0 is faster.
B_CROSSFADE	The parameter indicates a crossfade for mixing audio or video; 0 indicates that the the first of a pair of streams should be presented, 100 indicates that the other should be presented, and values in between indicate the degree of mixing that should occur.
B_COMPRESSION	The parameter indicates compression ratio; the value is 0 for no compression, 99 for 100:1 compression.
B_QUALITY	The parameter indicates quality level; the value is 0 for maximum compression, 100 for no compression.
B_BITRATE	The parameter indicates bit rate in bits per second.
B_GOP_SIZE	The parameter indicates a "group of pictures" such as how many frames apart a keyframe should be inserted into a video stream.
B_TUNER_CHANNEL	The parameter represents a channel tuner (like a TV channel); the value

	indicates the channel number.

The units of measurement are specified by *unit*, and the parameter's value can range from *minValue* to *maxValue*, with a granularity of *step*.

Name()

```
const char *Name(void)
```

Returns the name of the BParameterGroup. This name can be displayed to the user; for example, it could be used as the label of a BTab.

ParameterAt()

```
BParameter *ParameterAt(int32 index)
```

Returns the BParameter at the specified *index* within the BParameterGroup. If the *index* is negative, or is greater than `CountParameters()`-1, `NULL` is returned.

Web()

```
BParameterWeb *Web(void)
```

Returns the BParameterWeb that owns the group.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameterGroup Function Summary

Function Summary

Function	Description
CountGroups()	Returns the number of subgroups attached to the BParameterGroup
CountParameters()	Returns the number of parameters attached to the BParameterGroup
GroupAt()	Returns the BParameterGroup at the specified index
MakeContinuousParameter()	Creates a new BContinuousParameter and attaches it to the group
MakeDiscreteParameter()	Creates a new BDiscreteParameter and attaches it to the group
MakeNullParameter()	Creates a new BNullParameter and attaches it to the BParameterGroup
Name()	Returns the BParameterGroup's name
ParameterAt()	Returns the BParameter at the specified index
BParameterGroup()	Constructor
~BParameterGroup()	Destructor
Web()	Returns the BParameterWeb

	that owns the BParameterGroup

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNullParameter

Derived from: BParameter
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BNullParameter class represents a junction or I/O point in the parameter web. It doesn't take any value; instead, it's used primarily to identify special places in the parameter web that need to be displayed specially by the client application.

Typical uses of BNullParameters are to identify where signals enter or leave the node (input and output jacks, ADCs or DACs, and so forth).

Member Functions

ValueType()

```
virtual type_code ValueType(void)
```

Return a type_code indicating the data type of the control's value; this returns 0, since BNullParameters have no value.

The Be Book The Media Kit The Media Kit Index

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNullParameter Function Summary

Function Summary

Function	Description
ValueType()	Hook: return the type_code of the control's value

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BContinuousParameter

Derived from: BParameter
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BContinuousParameter class represents parameters whose values can vary along a continuous range. Examples include gain or equalizer controls, turntable pitch controls, and the like.

The default system theme implements BContinuousParameters as BSlider controls.

Call `BParameterGroup::MakeContinuousParameter()` to create a BContinuousParameter control.

Constructor and Destructor

You never create or delete a BContinuousParameter object yourself. Use the appropriate BParameterGroup functions to create these objects and add them to groups.

Member Functions

GetResponse() , SetResponse()

```
void GetResponse(response *response, float *factor, float *offset)
void SetResponse(response response, float factor, float offset)
```

`GetResponse()` returns in *response*, *factor*, and *offset* the response type, factor, and offset currently in effect for the BContinuousParameter.

`SetResponse()` sets these values.

The *response* describes how the BContinuousParameter's value is displayed to me the user. For example, if your parameter is polynomial in scale, you may wish to record only the base value, and let the BContinuousParameter handle presenting the value to the user as a polynomial by raising the value to the power specified by *factor* before displaying the value on the screen.

Constant	Description
B_LINEAR	For every unit of change in the parameter's value, the displayed value also changes by 1. The *factor* should be 1.
B_POLYNOMIAL	The *factor* is a power to which the control's value is raised before being displayed.
B_EXPONENTIAL	The *factor* is the base; the displayed value is the factor raised to the power of the parameter's value.
B_LOGARITHMIC	The *factor* is the base; the displayed value is the base *factor* logarithm of the parameter's value.

The *offset* is added to the value after the transformation specified by response is computed, but before displaying the value.

If you wish the displayed value to be v2+1, you would use the following call:

```
SetResponse(B_POLYNOMIAL, 2.0, 1.0);
```

The mapping specified by these functions is only used for controls that display the current value of the control. The default system theme doesn't currently display this information. Since this information is only used for display purposes, you will still have to handle (as appropriate for your needs) mapping between the value of the control and the parameter value it represents.

For example, a BContinuousParameter that selects a frequency between 20 Hz and 20,000 Hz might use a range from 0 to 3, with an exponent base of 10 and a multiplication factor of 20. In this case, the parameter will receive values from 0.0 to 3.0, which the UI control knows to map to the values 20 through 20,000 for display purposes.

MaxValue()

```
float MaxValue(void)
```

Returns the maximum possible value the BContinuousParameter can take, which was specified when the control was created. The value is in the units specified when the control was created, such as "dB" or "Hz."

MinValue()

```
float MinValue(void)
```

Returns the minimum possible value the BContinuousParameter can take, which was specified when the object was created. The value is in the units specified when the object was created, such as "dB" or "Hz."

SetResponse() see GetResponse()

ValueStep()

```
float ValueStep(void)
```

Returns the granularity of values between `MinValue()` and `MaxValue()` supported by the hardware.

ValueType()

```
virtual type_code ValueType(void)
```

Return a type_code indicating the data type of the control's value; by default, this is `B_FLOAT_TYPE`.

Constants

response

Declared in: be/media/ParameterWeb.h

Constant	Description
`B_LINEAR`	For every unit of change in the parameter's value, the displayed value also changes by 1. The *factor* should be 1.
`B_POLYNOMIAL`	The *factor* is a power to which the control's value is raised before being displayed.
`B_EXPONENTIAL`	The *factor* is the base; the displayed value is the factor raised to the power of the parameter's value.
`B_LOGARITHMIC`	The *factor* is the base; the displayed value is the base *factor* logarithm of the parameter's value.

These values define the response type for a BContinuousParameter.

`B_LINEAR` indicates that the user interface will display the parameter's value as-is.

`B_POLYNOMIAL` displays the parameter's value raised to the factor power. For instance, if *factor* is 2, the displayed value would be the parameter's value squared.

`B_EXPONENTIAL` displays the *factor* raised to the power of the parameter's value. If the parameter's value is 3 and the *factor* is 2, the displayed value would be 23 or 8.

`B_LOGARITHMIC` displays the base *factor* logarithm of the parameter's value; if the *factor* is 10 and the parameter's value is 3, the displayed value would be log10(3).

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

- BContinuousParameter

Function Summary

Function Summary

Function	Description
GetResponse()	Returns the response type
MaxValue()	Returns the maximum possible value of the BContinuousParameter
MinValue()	Returns the minimum possible value of the BContinuousParameter
ValueStep()	Returns the granularity of the BContinuousParameter
ValueType()	Hook: return the type_code of the control's value

---

The Be Book **The Media Kit** **The Media Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDiscreteParameter

Derived from: BParameter
Declared in: be/media/ParameterWeb.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

The BDiscreteParameter class represents a parameter that is set to one of a discrete set of values, and is usually used for non-linear values such as color spaces, multiplexers, and mutes or enables.

Each discrete value has a name that's displayed to the user by applications that draw a user interface for the node.

The default system theme renders these as BCheckBox controls, BRadioButton groups, or BMenuField controls, depending on the parameter kind. Other themes might implement these differently.

Constructor and Destructor

You never create or delete a BDiscreteParameter object yourself. Instead, call the **BParameterGroup::MakeDiscreteParameter()** function to create a BDiscreteParameter within the desired BParameterGroup.

Member Functions

AddItem()

```
status_t AddItem(int32 value, const char *name)
```

Adds a new *name/value* pair to the list of possible values the control can take. Whenever the selection named *name* is chosen, the user interface application will send a value change message to the control with the specified *value*.

RETURN CODES

- B_OK. The item was added successfully.
- B_NO_MEMORY. Not enough memory to add the item to the list.

CountItems()

```
int32 CountItems(void)
```

Returns the number of discrete values the control can take.

ItemNameAt() , ItemValueAt()

```
const char *ItemNameAt(int32 index)
int32 ItemValueAt(int32 index)
```

These functions return the name and value of the item at the specified *index*, where *index* ranges from 0 to CountItems()-1.

Selection values aren't necessarily in any sort of order, nor are they necessarily unique.

MakeEmpty()

```
void MakeEmpty(void)
```

Removes all items from the BDiscreteParameter.

MakeItemsFromInputs() , MakeItemsFromOutputs()

```
void MakeItemsFromInputs(void)
```

void **MakeItemsFromOutputs**(void)

These functions add all inputs or outputs as selections, where the input or output's name is used as the item's name, and the input or output's index number is used as the value of the item.

These shortcuts let you easily create the item lists for MUX-like controls, which allow the user to choose one of a number of inputs or outputs.

It's important to note that the correct inputs and outputs must already be connected to the parameter when you call either of these functions, otherwise no items will be added, since the inputs and outputs aren't there.

ValueType()

virtual type_code **ValueType**(void)

Returns `B_INT32_TYPE`, which is the data type a BDiscreteParameter's value is maintained in.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDiscreteParameter Function Summary

Function Summary

Function	Description
AddItem()	Adds a new item to the list of possible values
CountItems()	Returns the number of discrete values the control can take
ItemNameAt()	Returns the name of one of the available items
ItemValueAt()	Returns the value of one of the available items
MakeEmpty()	Removes all items
MakeItemsFromInputs()	Adds all inputs as items
MakeItemsFromOutputs()	Adds all outputs as items
ValueType()	Returns B_INT32_TYPE, the data type used by BDiscreteParameter objects

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTimeSource Function Summary

Function Summary

Function	Description
BroadcastTimeWarp()	Tells all slaved nodes about a jump in time
GetStartLatency()	Returns the time needed to start the time source
GetTime()	Returns the latest published time information
HandleMessage()	Dispatches a message received on the control port
IsRunning()	Reports whether or not the time source is running
Now()	Returns an approximation of what the current performance time is
PerformanceTimeFor()	Returns an approximation of the performance time corresponding to a particular real time
PublishTime()	Publishes the current time to the world at large
RealTime()	Returns the current absolute real time
RealTimeFor()	Returns an approximation of the real time corresponding to a particular performance time
Seek()	Hook: seek to the specified point in performance time

SendRunMode()	Informs slaved nodes of a run mode change
SetRunMode()	Hook: sets the node's run mode
SnoozeUntil()	Waits until the specified performance time
Start()	Hook: specifies a time at which the node should start
Stop()	Hook: stops the node
BTimeSource()	Constructor
TimeSourceOp()	Hook: handle a time source operation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Messaging

The Application Kit provides a message-passing system lets your application send messages to and receives messages from other applications (including the Be-defined servers and apps), and from other threads in your own application.

The primary messaging classes are:

- BMessage represents a message.
- BLooper runs a loop that receives in-coming messages and figures out which BHandler should handle them.
- BHandler defines hook functions that are called to handle in-coming messages.
- BMessenger represents a message's destination (a combination of BLooper and BHandler), whether it's local or remote. The object is most useful for sending messages to other applications—you don't need it for local calls.

The other messaging classes are:

- BMessageQueue is a fifo that holds a BLooper's in-coming messages.
- BMessageFilter is a device that can examine and (potentially) reject or re-route in-coming messages.
- BInvoker is a convenience class that lets you treat a message and its target (the BHandler that will handle the message) as a single object.
- BMessageRunner lets you send the same message over and over, at regular intervals.

The rest of this chapter looks at...

- The essential features of the four fundamental classes. ("Features of the Fundamental Classes")
- How a BLooper decides which BHandler should handle an in-coming message. ("From Looper to Handler")
- The different methods for sending messages and receiving replies. ("Sending a Message").

describes how the classes fit together in the messaging system with an emphasis on what you can do in your application to take part.

Features of the Fundamental Classes

Looked at collectively, the four fundamental messaging classes comprise a huge chunk of API. Fortunately, the essential part of this API is pretty small; that's what we're going to look at here.

The BMessage Class

In the BMessage class, there's one essential data member, and two essential functions:

- The **what** data member is an arbitrary **uint32** value that describes (symbolically) what the message is about. You can set and examine **what** directly—you don't have to use functions to get to it. The **what** value is called the object's *command constant*. The BeOS defines some number of command constants (such as **B_QUIT_REQUESTED**, and **B_MOUSE_DOWN**), but you'll also be creating constants of your own. Keep in mind that the constant's value is meaningless—it's just a code that identifies the "intent" of the message (and it's only meaningful if the receiver recognizes the constant).
- The two essential functions are **AddData()** and **FindData()**. These functions add data to a message you're about to send, and retrieve it from a message you just received. A BMessage can hold any amount of data; each data item (or "field") is identified by name, type, and index. For example, you can ask a message for the third boolean value named "IsEnabled" that it contains. In general, you use type-specific functions such as **Add/FindString()** and **Add/FindInt32()** rather than **Add/FindData()**. The query we just posed would actually look like this:

```
/* The args are: name, index, value (returned by reference) */
bool returnValue;
aMessage.FindBool("IsEnabled", 2, &returnValue);
```

In summary, a BMessage contains **(1)** a command constant and **(2)** a set of data fields. Every BMessage that's used in the messaging system must have a command constant, but not every object needs to have data fields. (Other parts of the BeOS use BMessages for their data only. The BClipboard object, for example, ignores a BMessage's command constant.)

When discussing system-generated BMessage objects, we refer to the object by its command constant. For example, "a **B_MOUSE_DOWN**" means "a BMessage that has **B_MOUSE_DOWN** as its command constant".

Notice that a BMessage doesn't know how to send itself. However, as we'll see later, it does know how to reply to its sender once it's in the hands of the recipient.

The BLooper Class

BLooper's role is to receive messages and figure out what to do with them. There are four parts to this job, embodied in these functions:

- Every BLooper spawns a thread in which it runs a message loop. It's in this thread that the object receives messages. But you have to kick the BLooper to get it to run; you do this by calling the **Run()** function. When you're done with the obejct—when you no longer need it to receive messages—you call **Quit()**.

- Although you never invoke it directly, **DispatchMessage()** is the guts of the message loop. All messages that the looper receives show up in individual invocations of **DispatchMessage()**. The function decides where the message should go next, which is mostly a matter of deciding whether **(1)** the message should be handled by a system-defined hook function, or **(2)** passed to BHandler's **MessageReceived()** function (which we'll talk about in a moment). Three other important aspects of **DispatchMessage()** are...

- It runs in the BLooper's message thread (or *message loop*); this is a separate thread that the object spawns specifically to receive in-coming messages.

- Individual **DispatchMessage()** invocations are synchronous with regard to the loop. In other words, when a message shows up, **DispatchMessage()** is called and runs to completion before the next message can be processed. (Messages that show up while **DispatchMessage()** is busy aren't lost—they're queued up in a BMessageQueue object.)

- It's fully implemented by BLooper (and kit classes derived from BLooper). You should rarely need to override it in your application.

- The **PostMessage()** function delivers a message to the BLooper. In other words, it invokes **DispatchMessage()** in the looper's message thread. You call **PostMessage()** directly in your code. For example, here we create a BMessage and post it to our BApplication object (BApplication inherits from BLooper):

```
/* This form of the BMessage constructor sets the command constant. */
BMessage *myMessage = new BMessage(YOUR_CONSTANT_HERE);
be_app->PostMessage(myMessage)
```

The BApplication and BWindow classes inherit from BLooper.

The BHandler Class

BHandler objects are called upon to handle the messages that a BLooper receives. A BHandler depends on two essential function:

- **MessageReceived()** is the function that a BLooper calls to dispatch an in-coming message to the BHandler (the BMessage is passed as the function's only argument). This is a hook function that a BHandler subclass implements to handle the different types of messages that it expects to receive. Most implementations examine the message's command constant and go from there. A typical outline looks like this:

```
void MyHandler::MessageReceived(BMessage *message)
{
   /* Examine the command constant. */
   switch ( message->what ) {
```

```
   case YOUR_CONSTANT_HERE:
      /* Call a function that handles this sort of message. */
      HandlerFunctionA();
      break;

   case ANOTHER_CONSTANT_HERE:
      /* ditto */
      HandlerFunctionB();
      break;

   default:
      /* Messages that your class doesn't recognize must be passed
      * on to the base class implementation. */
      baseClass::MessageReceived(message);
      break;
   }
}
```

- BHandler's other essential function is defined by BLooper: **`BLooper::AddHandler()`**. This function adds the (argument) BHandler object to the (invoked-upon) BLooper's list of candidate handlers (its *handler chain*). If a BHandler wants to handle messages that are received by a BLooper, it must first be added to the BLooper's handler chain.

BLooper inherits from BHandler, and automatically adds itself to its own handler chain. This means that a BLooper can handle the messages that it receives—this is the default behaviour for most messages. We'll examine this issue in depth later in this chapter.

The other classes that inherit from BHandler are BView and BShelf (both in the Interface Kit).

The BMessenger Class

A BMessenger's most important feature is that it can send a message to a remote application. To do this takes two steps, which point out the class' essential features:

- You identify the application that you want to send a message to (the "target") in the BMessenger constructor. An application is identified by its *app signature* (a MIME string).
- The **`SendMessage()`** function sends a message to the target.

BMessengers can also be used to target local looper/handler pairs.

From Looper to Handler

A BLooper pops a message from its message queue and, within its `DispatchMessage()` function, dispatches the message by invoking a BHandler function. But **(1)** which BHandler and **(2)** which function?

Finding a Handler

First, let's answer the "which BHandler" question. To determine which BHandler should handle an in-coming message, a BLooper follows these steps:

1. *Does the BMessage target a specific BHandler?* Both the `BLooper::PostMessage()` and `BMessenger::SendMessage()` functions provide additional arguments that let you specify a *target handler* that you want to have handle the message (you can also set the target in the BMessenger constructor). If a BHandler is specified, the BMessage will show up in that object's `MessageReceived()` function (or it will invoke a message-mapped hook function, as explained below).

2. *Does the BLooper designate a preferred handler?* Through its `SetPreferredHandler()` function, a BLooper can designate one of the objects in its handler chain as its *preferred handler*, and passes all messages to that object.

3. *The BLooper handles the BMessage itself*. If there's no target handler or preferred handler designation, the BLooper handles the message itself—in other words, the message is passed to the BLooper's own `MessageReceived()` function (or message-mapped hook).

We should mention here that both the BApplication and the BWindow class fine-tune this decision process a bit. However, the meddling that they do only applies to system messages (described below). The messages that you define yourself (i.e. the command constants that your application defines) will always follow the message dispatching path described here.

If you look at the `DispatchMessage()` protocol, you'll notice that it has a BMessage and a BHandler as arguments. In other words, the "which handler" decision described here is actually made before `DispatchMessage()` is called. In general, this is an implementation detail that you shouldn't worry about. If you want to think that `DispatchMessage()` makes the decision—and we've done nothing to disabuse you of this notion—go ahead and think it.

Finding a Function

As described above, a BLooper passes a BMessage to a BHandler by invoking the latter's `MessageReceived()` function. This is true of all messages that you create yourself, but it isn't true of certain messages that the system defines and sends. These system-generated messages (or *system messages*)—particularly those that report user events such as `B_MOUSE_DOWN` or `B_APP_ACTIVATED`—invoke specific hook functions.

For example, when the user presses a key, a `B_KEY_DOWN` message is sent to the active BWindow object.. From within its `DispatchMessage()` function, BWindow invokes the `MouseDown()` function

of the BView that currently holds *keyboard focus*. (When a BView is made the focus of keyboard events, its window promotes it to preferred handler.)

So the question of "which function" is fairly simple: If the BMessage is a system message that's mapped to a hook function, the hook function is invoked. If it's not mapped to a hook function, the BHandler's **MessageReceived()** function is invoked.

A full list of system messages and the hook functions that they're mapped to is given in the System Messages Appendix. Note that not all system messages are mapped to hook functions; some of them do show up in **MessageReceived()**.

Inheritance and the Handler Chain

Let's look at **MessageReceived()** again. It was asserted, in a code snippet shown earlier, that a typical **MessageReceived()** implementation should include an invocation of the base class' version of the function:

```
void MyHandler::MessageReceived(BMessage *message)
{
   switch ( message->what ) {

   /* Command constants that you handle go here. */

   default:
      /* Messages that your class doesn't recognize must be passed
      * on to the base class implementation. */
      baseClass::MessageReceived(message);
      break;
   }
}
```

This isn't just a good idea—it's an essential part of the messaging system. Forwarding the message to the base class does two things: It lets messages **(1)** pass up the class hierarchy, and **(2)** pass along the handler chain (in that order).

Passing up the class hierarchy is mostly straight-forward—it's no different for the **MessageReceived()** function than it is for any other function. But what happens at the top of the hierarchy—at the BHandler class itself—adds a small wrinkle. BHandler's implementation of **MessageReceived()** looks for the next handler in the BLooper's handler chain and invokes that object's **MessageReceived()** function.

Sending a Message

There are two functions that send messages to distinct recipients:

- **BLooper::PostMessage()** can be used if the target (the BLooper that the **PostMessage()** function is invoked upon) lives in the same application as the message sender.
- **BMessenger::SendMessage()** lets you send messages to remote applications. The BMessenger object acts as a proxy for the remote app. (**SendMessage()** can also be used to send a message to a local BLooper, for reasons that we'll discuss later.)

The PostMessage() Function

You can post a message if the recipient BLooper is in your application:

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), targetHandler);
```

As shown here, you can specify the handler that you want to have handle a posted message. The only requirement is that the BHandler must belong to the BLooper.

If the handler argument is NULL, the message is handled by the looper's *preferred handler*

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), NULL);
```

By using the default handler, you let the looper decide who should handle the message.

The creator of the BMessage retains ownership and is responsible for deleting it when it's no longer needed.

The SendMessage() Function

If you want to send a message to another application, you have to use BMessenger's **SendMessage()** function. First, you construct a BMessenger object that identifies the remote app by signature...

```
BMessenger messenger("application/x-some-app");
```

...and then you invoke **SendMessage()**:

```
messenger.SendMessage(new BMessage(DO_SOMETHING));
```

The creator of the BMessage retains ownership and is responsible for deleting it when it's no longer needed.

Handling a Reply

Every BMessage that you send identifies the application from which it was sent. The recipient of the message can reply to the message whether you (the sender) expect a reply or not. By default, reply messages are handled by your BApplication object. If you want reply messages to be handled by some other BHandler, you specify the object as a final argument to the **PostMesssage()** or **SendMessage()** call:

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), targetHandler, replyHandler);
/* and */
myMessenger.SendMessage(&message, replyHandler);
```

The reply is sent asynchronously with regard to the PostMessage()/SendMessage() function.

SendMessage() (only) lets you ask for a reply message that's handed back synchronously in the SendMessage() call itself:

```
BMessage reply;
myMessenger.SendMessage(&message, &reply);
```

SendMessage() doesn't return until a reply is received. A default message is created and returned if the recipient doesn't respond quickly enough.

Receiving a Message

BMessage's **SendReply()** function has the same syntax as **SendMessage()**, so it's possible to ask for a synchronous reply to a message that is itself a reply,

```
BMessage message(READY);
BMessage reply;
theMessage->SendReply(&message, &reply);
if ( reply->what != B_NO_REPLY ) {
   . . .
}
```

or to designate a BHandler for an asynchronous reply to the reply:

```
theMessage->SendReply(&message, someHandler);
```

In this way, two applications can maintain an ongoing exchange of messages.

Handler Associations

To be notified of an arriving message, a BHandler must "belong" to the BLooper; it must have been added to the BLooper's list of eligible handlers. The list can contain any number of objects, but at any given time a BHandler can belong to only one BLooper.

Handlers that belong to the same BLooper can be chained in a linked list. If an object can't respond to a message, the system passes the message to its next handler.

BLooper's **`AddHandler()`** function sets up the looper-handler association; BHandler's **`SetNextHandler()`** sets the handler-handler link.

Message Filters

The BMessageFilter class lets create filtering functions that examine and re-route (or reject) incoming messages before they're processed by a BLooper. Message filters can also be applied to individual BHandler objects.

Message Protocols

Both the source and the destination of a message must agree upon its format—the command constant and the names and types of data fields. They must also agree on details of the exchange—when the message can be sent, whether it requires a response, what the format of the reply should be, what it means if an expected data item is omitted, and so on.

None of this is a problem for messages that are used only within an application; the application developer can keep track of the details. However, protocols must be published for messages that communicate between applications. You're urged to publish the specifications for all messages your application is willing to accept from outside sources and for all those that it can package for delivery to other applications.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageQueue

Derived from: *none*
Declared in: be/app/MessageQueue.h
Library: libbe.so
Summary: *more...*

The BMessageQueue class completes the implementation of BLooper by providing a first-in/first-out stack in which the looper can place in-coming BMessages. In general, the message dispatching mechanism of BLooper should suffice. However, if you ever need to manipulate a BMessage queue directly, you can do so.

Constructor and Destructor

BMessageQueue()

```
BMessageQueue(void)
```

Creates an empty BMessageQueue object.

~BMessageQueue()

```
virtual ~BMessageQueue()
```

Deletes all the objects in the queue and all the data structures used to manage the queue.

Member Functions

AddMessage() , RemoveMessage()

void **AddMessage(**BMessage **message***)**
void **RemoveMessage(**BMessage **message***)**

`AddMessage()` adds *message* to the far end of the queue. `RemoveMessage()` removes a particular *message* from the queue and deletes it.

CountMessages() , IsEmpty()

int32 **CountMessages(**void**)** const
bool **IsEmpty(**void**)** const

`CountMessages()` returns the number of messages currently in the queue.

`IsEmpty()` returns `true` if the object doesn't contain any messages, and `false` otherwise.

FindMessage()

BMessage ***FindMessage(**int32 *index***)** const BMessage ***FindMessage(**uint32 *what*, int32 *index* = 0**)** const

`FindMessage()` returns a pointer to the *index*'th BMessage in the queue, where index 0 signifies the message that's been in the queue the longest. The second version lets you specify a *what* field value; in this case, only messages that match the *what* argument are counted. If no message matches the criteria, the functions return `NULL`.

The message is not removed from the message queue.

IsEmpty() see CountMessages()

Lock() , Unlock()

bool **Lock(**void**)**
void **Unlock(**void**)**

These functions lock and unlock the BMessageQueue, so that another thread won't alter the contents of the queue while it's being read. **`Lock()`** doesn't return until it has the queue locked;

it always returns `true`. **`Unlock()`** releases the lock so that someone else can lock it. Calls to these functions can be nested.

See also: **`BLooper::Lock()`**

NextMessage()

BMessage ***NextMessage**(void)*

Removes and returns the oldest message from the queue. If the queue is empty, the function returns `NULL`.

See also: **`FindMessage()`**

RemoveMessage() see AddMessage()

Unlock() see Lock()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageQueue Function Summary

Function Summary

Function	Description
AddMessage()	Adds a message to the end of the queue
CountMessages()	Returns the number of messages currently in the queue
FindMessage()	Locates the message in the queue at a specific index
IsEmpty()	Tests whether the queue is empty
Lock()	Locks the queue
BMessageQueue()	Constructor
~BMessageQueue()	Destructor
NextMessage()	Removes the oldest message from the queue and returns it
RemoveMessage()	Removes a given message from the queue
Unlock()	Unlocks the queue

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Index C Functions

Declared in: be/kernel/fs_index.h
Library: libroot.so
Summary: *more...*

Before a query can be performed on a file system, indices need to be established in which to search. An index contains a list of files that can be searched given a particular attribute and the desired value.

An index directory is, essentially, a list of the indices on a particular disk; it keeps track of the attributes that can be searched upon. Each disk has its own index directory. For instance, a disk might have the following indices in its index directory:

- MAIL:subject
- MAIL:from
- MAIL:to
- MAIL:priority

Each of these is the name of an index that can be queried using either the **`fs_query`** API or the BQuery class.

For example, if an e-mail program wishes to allow searching on the name of the sender of a message, it needs to create an index for the "sender" attribute. Once this index is established, any file that has the "sender" attribute added to it will be added to the "sender" index.

Files which had the "sender" attribute attached *before* the "sender" index was created, however, will *not* be in the index until their "sender" attribute is updated, at which time they will be added to the index. For this reason, you should consider installing your indices when your application is installed, or when it is initially launched.

Since each disk has its own index directory, if you want all disks to have your indices available, you need to create them on each device. You can do this by using the functions described here in conjunction with the **`fs_info`** functions.

There are three indices that are present on every disk:

Index	Meaning
name	The name of the file.
size	The size, in bytes, of the file.
last_modified	The date the file was last changed.

You can always perform queries in these indices. Their names are reserved; you can't create or remove indices by these names.

The Index Directory

The following sample function opens the index directory for a specified device and, in a loop, reads every entry, printing their names to standard output. This presents a list of each index available on the disk.

```
void ListIndex(int32 device)
{
   DIR *d;
   index_info info;
   struct dirent *ent;

   d = fs_open_index_dir(device);
   if (!d) {
      fprintf(stderr, "Unable to open index.n");
      return;
   }

   while (ent = fs_read_index_dir(d)) {
      printf("%sn", ent->d_name);
   }
   fs_close_index_dir(d);
}
```

After calling **fs_open_index_dir()** to open the index directory for the device passed into the function (and handling the error that might occur if that function fails), a **while** loop iterates through all the entries in the directory, calling **fs_read_index_dir()** to obtain the desired information, then printing that information to standard output.

When there are no indices left, **fs_read_index_dir()** returns **NULL** and the **while** loop exits. At this point, **fs_close_index_dir()** is called to close the index directory.

Installing and Removing Indices

To install a new index, use the **fs_create_index()** function. For example, to create an index for the attribute "GOLF:Handicap" on device 4 you would do this:

```
fs_create_index(4, "GOLF:Handicap", B_INT16_TYPE, 0);
```

This creates an empty index for the golf handicap attribute, which is stored as a 16-bit integer. Once the index has been created, any file that gets a "GOLF:Handicap" attribute added or changed will be indexed.

(You usually shouldn't hard-code a device number, of course; you can obtain a device number for a specific disk by using the **fs_info** functions or **stat()**.)

If you later need to remove the index, you can do so by calling **fs_remove_index()**, like this:

```
fs_remove_index(4, "GOLF:Handicap");
```

You should be careful when deciding to delete an index. If the user still has files around that contain indexed attributes that they want to be able to search for—using the Find panel in the Tracker, for example—they will not be able to do so after the index has been removed. So you need to decide when it is appropriate to remove indices; the choice is yours, but choose wisely, or you might annoy users.

Index Functions

fs_close_index_dir()

```
int fs_close_index_dir(DIR *dirp)
```

Closes the specified index directory and frees *dirp*. You should pass into this function the pointer returned from a previous **fs_open_index_dir**().

You should always use this function to close index directories after you finish using them.

If successful, this function returns 0; otherwise it returns -1and sets **errno** to one of the following codes.

- **B_FILE_ERROR**. Invalid directory reference specified.
- **B_BAD_FILE**. Invalid directory reference specified.

fs_create_index()

```
int fs_create_index(dev_t device,
    const char *name,
    int type,
    uint flags)
```

Creates a new index called *name* on the specified *device*. Once this has been done, adding an attribute named *name* to a file causes the file to be added to the *name* index, such that subsequent queries will be able to search for files that contain the *name* attribute.

type indicates the kind of data the attribute will contain. Standard types are defined in the **be/support/TypeConstants.h** header file. Different file systems may support different types. BFS supports the following types of attributes:

- **B_INT32_TYPE**
- **B_INT64_TYPE**
- **B_FLOAT_TYPE**
- **B_DOUBLE_TYPE**
- **B_STRING_TYPE**
- **B_MIME_STRING_TYPE**

flags is currently unused and should always be 0.

If successful, this function returns 0; otherwise it returns -1and sets **errno** to one of the following codes.

- **B_BAD_VALUE**. The device does not exist, or *name* is reserved.
- **B_NOT_ALLOWED**. The device is read-only.
- **B_NO_MEMORY**. Insufficient memory to complete the operation.
- **B_FILE_EXISTS**. The index *name* already exists.
- **B_DEVICE_FULL**. There's not enough room on the device to create the index.
- **B_FILE_ERROR**. Invalid directory reference.
- **B_ERROR**. The index type passed isn't supported.

---

fs_open_index_dir()

DIR ***fs_open_index_dir**(dev_t *device*)

Opens the index directory for the volume identified by *device*. Once open, you can retrieve the names of the indices on the volume by calling **fs_read_index_dir()**.

When you have finished using the index directory, call **fs_close_index_dir()** to close it.

If the index directory is opened successfully, a pointer to a directory structure is returned. This pointer should be passed to the other **fs_index** functions to read entries from the index directory, as well as to close the directory when you're finished with it.

If an error occurs while opening the index directory, this function returns **NULL** and sets **errno** to an appropriate value.

- **B_FILE_ERROR**. A file error prevented the operation.
- **B_BAD_VALUE**. Invalid device number specified.
- **B_LINK_LIMIT**. A cyclic loop was detected in the directory structure.
- **B_NO_MEMORY**. Insufficient memory to complete the operation.
- **B_BUSY**. The specified file is currently in use.
- **B_NO_MORE_FDS**. Too many open files; all file descriptors are in use.

---

fs_read_index_dir()

struct dirent ***fs_read_index_dir**(DIR **dirp*)

Reads the current entry from the open index directory referenced by *dirp*, and bumps the pointer to point to the next entry. dirp should have been obtained through a previous call to `fs_open_index_dir()`. The returned **dirent** pointer contains information about the index entry, including the name of the attribute represented by the index. This pointer belongs to the system; you must not delete it.

Through repeated calls to `fs_read_index_dir()`, you can obtain a list of all the indices available on the device. When you reach the end of the list, errno is set to `B_ENTRY_NOT_FOUND`.

If an error occurs, this function returns **NULL** and sets **errno** to an appropriate value.

- `B_FILE_ERROR`. Invalid file descriptor *fd* specified.
- `B_NOT_A_DIRECTORY`. *dirp* does not represent a valid index directory.
- `B_ENTRY_NOT_FOUND`. You've reached the end of the list.

---

fs_remove_index()

int **fs_remove_index(**dev_t *device*, const char ***index_name*)**

Deletes the index named *index_name* from the specified *device*. Once the index is deleted, it will no longer be possible to use the query system to search for files with the corresponding attribute.

Use this function to remove an index that you no longer wish or need to be able to search upon. For example, if your application is being uninstalled by your user-friendly uninstaller program, and it's no longer meaningful to be able to search on a given attribute, you should use this function to delete the index for that attribute.

You should be careful when deciding whether or not to delete an index, however. If the user still has files around that they want to be able to search, using the Tracker's Find panel, for instance, and you've deleted the index for that attribute, they'll be most displeased. There's a grey area you need to wade through in determining whether or not to delete your indices; your decision needs to be based on the specifics of what your application does and how it will be used.

If the index is removed successfully, `fs_remove_index()` returns 0. Otherwise, it returns -1 and sets **errno** to an appropriate value.

- `B_FILE_ERROR`. A file system error prevented the operation.
- `B_BAD_VALUE`. Invalid device number specified.
- `B_NOT_ALLOWED`. Can't remove a system-reserved index ("name," "size," "last_modified"), or the device is read-only.
- `B_NO_MEMORY`. Insufficient memory to complete the operation.
- `B_ENTRY_NOT_FOUND`. The specified index does not exist.

fs_rewind_index_dir()

```
int fs_rewind_index_dir(DIR *dirp)
```

Rewinds the specified index directory to the beginning of its list of indices. This allows you to start over again at the top of a device's index directory and make your way down toward the bottom.

Returns a result code specifying whether or not the operation was successful.

Unlike the most of the other file system functions, `fs_rewind_index_dir()` *doesn't* set **errno**.

RETURN CODES

- `B_OK`. Success.
- `B_FILE_ERROR`. Invalid directory reference specified.

---

fs_stat_index()

```
int fs_stat_index(dev_t device,
    const char *index_name,
    struct index_info *info)
```

Returns, in the `index_info` structure pointed to by *info*, information about the index named *index_name* on the specified *device*.

The `index_info` structure is defined as follows:

```
typedef struct index_info
{
   uint32      type;
   off_t       size;
   time_t      modification_time;
   time_t      creation_time;
   uid_t       uid;
   gid_t       gid;
}
```

type contains a code defining the format of the data contained by the attribute represented by the index; standard values for this field are defined in the **be/support/TypeConstants.h** header file.

- `size` specifies the size of the data the attribute contains.

- `modification_time` contains the time the index was last changed, in seconds since January 1, 1970.
- `creation_time` contains the date and time the index was originally created, in seconds since January 1, 1970.
- `uid` contains the user ID of the owner of the index.
- `gid` contains the group ID of the owner of the index.

If the function is successful, it returns 0; otherwise, it returns -1 and sets errno to an appropriate value

-
- **`B_ENTRY_NOT_FOUND`**. The requested index was not found.
- **`B_BAD_VALUE`**. Invalid device number specified.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Index C Functions Function Summary

Function Summary

Function	Description
fs_close_index_dir()	Closes the index directory
fs_create_index()	Creates a new index on the device
fs_open_index_dir()	Opens the index directory for a device
fs_read_index_dir()	Reads the next entry in the index directory
fs_remove_index()	Removes an index from the device
fs_rewind_index_dir()	Resets the index directory iterator
fs_stat_index()	Returns information about the specified index

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSymLink

Derived from: BNode
Declared in: be/storage/SymLink.h
Library: libbe.so
Summary: *more...*

A "symbolic link" or *symlink* is a file that "points to" some other entry. The pointed-to (or, better, "linked-to") entry can be a plain file, directory, or another symlink (which links to yet another entry, and so on). Furthermore, the entry can be abstract—you can create a symlink to an entry that doesn't exist.

The data in a symlink is the pathname to the linked-to entry. The pathname can be absolute or relative. If it's relative, the linked-to entry is found by reckoning the pathname of off the directory in which the symlink lives. Relative pathnames can contain "." and ".." components.

The thing to keep in mind, when dealing with symlinks, is that they link to *entries*, not *nodes*. If you link a symlink to an (existing) entry named "/boot/home/fido" and then the user moves **fido** to **rover** (or deletes **fido**), the symlink is not updated. It will still link to "/boot/home/fido".

Furthermore, symlinks that contain relative pathnames have a further "problem": Let's say you create a symlink in **/boot/home** that links to "fido". If you move the symlink to some other directory, it will link to the entry named "fido" in the new directory.

The BSymLink class creates objects that know how to read a symlink's data. The class does not create new symlinks; to create a symlink, you use BDirectory's `CreateSymLink()` function.

BSymLink objects are no smarter than the symlinks files themselves. For example, BSymLinks can't resolve the fido/rover "problem".

The only really useful BSymLink function is `ReadLink()`. This function returns the data that the symlink contains. The other functions are convenient, but they're not essential.

Initialization and File Descriptors

When you initialize a BSymLink object, you pass in a pathname or `entry_ref` (or whatever) that refers to an existing symlink. The BSymLink object then represents that symlink—it doesn't represent the (node of the) linked-to entry. Furthermore, you can't ask a BSymLink to "resolve itself"—it can't pass you back a BEntry object that represents the linked-to entry.

If you want the BEntry of the linked-to entry, simply initialize a BEntry object with the ref (or whatever) to the symlink and tell it to traverse (set the trailing argument to **`true`**).

For example, in the following code, `link` is a BSymLink to the symlink "/boot/home/fidoLink" and `entry` is a BEntry to the entry that the symlink links-to:

```
BSymLink link("/boot/home/fidoLink");
BEntry entry("/boot/home/fidoLink", true);
```

Like all nodes, BSymLink allocates a file descriptor. Remember, this is a file descriptor that's open on the symlink node itself, not the (node of the) linked-to entry.

Constructor and Destructor

BSymLink()

```
BSymLink(void)
BSymLink(const BSymLink &link)
BSymLink(const entry_ref *ref)
BSymLink(const BEntry *entry)
BSymLink(const char *path)
BSymLink(const BDirectory *dir, const char *path)
```

Creates a new BSymLink object, initializes it according to the arguments, and sets `InitCheck()` to return the status of the initialization.

- The default constructor does nothing and sets `InitCheck()` to `B_NO_INIT`. To initialize the object, call `SetTo()`.
- The copy constructor creates a new BSymLink that's open on the same node as that of the argument.
- For information on the other constructors, see the analogous `SetTo()` functions in the BNode class; BSymLink inherits them without change.

~BSymLink()

```
virtual ~BSymLink()
```

Closes the object's file descriptor and destroys the object.

Member Functions

IsAbsolute()

bool **IsAbsolute(**void**)**

Returns true if the symlink contains an absolute pathname.

MakeLinkedPath()

ssize_t **MakeLinkedPath(**const BDirectory **dir*, BPath **path***)** const
ssize_t **MakeLinkedPath(**const char **dirPath*, BPath **path***)** const

This function creates an absolute pathname to the linked-to entry and returns it as a BPath object. For this to work you have to tell the object which directory you want to reckon off of (in case the symlink specifies a relative path). This should be the directory in which the symlink itself lives.

Remember—a BSymLink is a node, and nodes don't know what directory they live in. That's why you have to tell it here.

If the symlink contains an absolute path, then the *dir* or *dirPath* arguments are ignored. Nonetheless, they must be supplied.

The function returns the length of the pathname that's set in BPath (or an error).

RETURN CODES

- **B_FILE_ERROR**. The object is uninitialized.
- **B_BAD_VALUE**. The object doesn't refer to a symlink, or *dir*/*dirPath* is NULL.
- **B_NAME_TOO_LONG**. They concatenated pathname is too long.

ReadLink()

ssize_t **ReadLink(**char **buf*, size_t *length***)**

Copies the contents of the symlink into *buf*. *length* is the size of the buffer; to be perfectly safe, the buffer should be **B_PATH_NAME_LENGTH** characters long. The function returns the number of bytes that were copied (or it returns an error).

The symlink's contents is the pathname (relative or absolute) to the linked-to entry. Note that since the pathname might be relative, **ReadLink()** can't give you a BPath object. If you want a

BPath to the linked-to entry, see `MakeLinkedPath()`.

RETURN CODES

- `B_FILE_ERROR`. The object is uninitialized.
- `B_BAD_VALUE`. The object doesn't refer to a symlink.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSymLink Function Summary

Function Summary

Function	Description
IsAbsolute()	Tests if the symbolic link contains an absolute pathname
MakeLinkedPath()	Returns an absolute pathname to the linked-to entry
ReadLink()	Reads the contents of the symbolic link
BSymLink()	Constructor
~BSymLink()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFile Function Summary

Function Summary

Function	Description
=	Initializes the object to refer to the same file as another
BFile()	Constructor
~BFile()	Destructor
GetSize()	Returns the size of the file's data portion
InitCheck()	Returns the status of the most recent initialization
IsReadable()	Tests if the BFile was initialized to read from the file
IsWritable()	Tests if the BFile was initialized to write to the file
Position()	Returns the location of the file pointer
ReadAt()	Reads from an arbitrary file position
Read()	Reads from the current file position
Seek()	Sets the location of the file pointer
SetSize()	Sets the size of the file's data portion
SetTo()	Initializes the object to another file
Unset()	Uninitializes the object
WriteAt()	Writes to an arbitrary file position
Write()	Writes to the current file position

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BVolumeRoster

Derived from: *none*
Declared in: be/storage/VolumeRoster.h
Library: libbe.so
Summary: *more...*

The BVolumeRoster class keeps track of the volumes that are mounted in the file system hierarchy. It lets you know about volumes in two ways:

- It lists the volumes that are currently mounted. You can step through the list through iterative calls to the `GetNextVolume()` function.
- It lets you know when new volumes are mounted, and when existing volumes are unmounted. (See `StartWatching()`.)

How you create your BVolumeRoster object depends on what you're going to do with it:

- If you simply want to step through the volume list, then creating a BVolumeRoster on the stack is sufficient.
- However, if you want to watch for volumes being mounted and unmounted, then you *must* keep your BVolumeRoster object around. The watching stops when the object is deleted.

A single BVolumeRoster object can perform both functions: You can use it to step through the volume list at the same time that it's watching volumes.

Constructor and Destructor

BVolumeRoster()

```
BVolumeRoster(void)
```

Creates a new BVolumeRoster object. You don't have to "initialize" the object before using it (as you do with most other Storage Kit classes). You can call `GetNextVolume()` (or whatever) immediately after constructing.

~BVolumeRoster()

```
virtual ~BVolumeRoster()
```

Destroys the object. If this BVolumeRoster object was watching volumes, the watch is called off.

Member Functions

GetBootVolume()

status_t **GetBootVolume**(BVolume **boot_vol*)

Initializes *boot_vol* to refer to the "boot volume." This is the volume that was used to boot the computer. *boot_vol* must be allocated before you pass it in. If the boot volume can't be found, the argument is uninitialized.

(Currently, this function looks for the volume that's mounted at **/boot**. The only way to fool the system into thinking that there isn't a boot volume is to rename **/boot**—not a smart thing to do.)

RETURN CODES

- **B_NO_ERROR**. The boot volume was successfully retrieved.
- **B_ENTRY_NOT_FOUND**. The boot volume wasn't found.

GetNextVolume() , Rewind()

status_t **GetNextVolume**(BVolume **volume*)
void **Rewind**(void)

`GetNextVolume()` retrieves the "next" volume from the volume list and uses it to initialize the argument (which must be allocated). When the function return **B_BAD_VALUE**, you've reached the end of the list.

`Rewind()` rewinds the volume list such that the next `GetNextVolume()` will return the first element in the list.

RETURN CODES

- **B_NO_ERROR**. The next volume was successfully retrieved.
- **B_BAD_VALUE**. You've reached the end of the volume list.

StartWatching() , StopWatching() , Messenger()

status_t **StartWatching**(BMessenger *messenger* = be_app_messenger)
void **StopWatching**(void)
BMessenger **Messenger**(void) const

These functions start and stop the BVolumeRoster's volume-watching facility. (This is actually just a convenient cover for the Node Monitor.)

- `StartWatching()` registers a request for notifications of volume mounts and unmounts. The notifications are sent (as BMessages) to the BHandler/BLooper pair specified by the argument. There are separate messages for mounting and unmounting; their formats are described below. The caller retains possession of the BHandler/BLooper that the BMessenger represents. The volume watching continues until this BVolumeRoster object is destroyed, or until you call...
- `StopWatching()`. This function tells the volume-watcher to stop watching. In other words, notifications of volume mounts and unmounts are no longer sent to the BVolumeRoster's target.
- `Messenger()` returns a copy of the BMessenger object that was set in the previous `StartWatching()` call.

There are separate notifications (BMessages) for volume-mounted and volume-unmounted events. See the B_DEVICE_MOUNTED and B_DEVICE_UNMOUNTED descriptions in "The Node Monitor" section of this chapter.

RETURN CODES

- **`B_NO_ERROR`**. The volume-watcher was successfully started or stopped.
- **`B_BAD_VALUE`**. Poorly formed BMessenger.
- **`B_NO_MEMORY`**. Couldn't allocate resources.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BVolumeRoster Function Summary

Function Summary

Function	Description
GetBootVolume()	Initializes a BVolume to the boot volume
GetNextVolume()	Retries the next volume from the volume list
Messenger()	Returns a BMessenger to the target of volume notification messages
Rewind()	Resets the volume list iterator
StartWatching()	Begin receiving volume mount and unmount notifications
StopWatching()	Stop receiving volume mount and unmount notifications
BVolumeRoster()	Constructor
~BVolumeRoster()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BEntry Function Summary

Function Summary

Function	Description
!=	Tests if two BEntry objects refer to different entries
=	Initializes an object to refer to the same entry as another
==	Tests if two BEntry objects refer to the same entry
BEntry()	Constructor
Exists()	Tests whether the entry exists
GetName()	Returns the leaf name of the entry
GetParent()	Returns the directory in which the entry lives
GetPath()	Returns the full pathname of the entry
GetRef()	Gets the entry_ref for the object's entry
GetStat()	Gets the stat structure for the entry
get_ref_for_path()	Returns an entry_ref for the specified pathname
InitCheck()	Returns the status of the last initialization
MoveTo()	Moves the entry to a new location
Remove()	Removes the entry from the directory
Rename()	Moves the entry to a new location
SetTo()	Initializes the object to refer to another entry

Unset()	Uninitializes the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

File System Architecture

All file systems that the BeOS recognizes are organized hierarchically. Most file systems use hierarchical organization naturally; any other type of organization must be adapted at the "file system handler" level to simulate a hierarchy (in order to be recognized by the BeOS). The result is that all file systems can be treated similarly in terms of their organizational architecture.

Hierarchical Organization

The tenets of a Be-recognized hierarchical file system are these:

- Data is organized as a tree of directories and files.
- Directories contain files.
- Each file is a member of only one directory (at a time). The directory is said to be the file's "parent." Once hard links are supported (as explained below) the one-parent rule will no longer hold.
- Similarly, each directory has a single parent (directory).
- The hierarchy "fans out" from the file system's *root directory*. The root directory is the common ancestor for all files and directories in the hierarchy.
- Every file and directory has a "leaf" name; leaf names must be unique within the containing directory.
- The only illegal characters in a leaf name (in the Be File System) are "/" and NUL (the character with the numeric value 0x00). Also, every directory automatically contains entries named "." and "..".

In the BeOS, more than one file system can be mounted at a time. Each distinct file system is mounted within the *root file system* (described in the next section). Because of this, all (non-root) file systems maintain distinct boundaries. However, these otherwise-tidy boundaries are smudged a bit by...

Symbolic Links

The BeOS recognizes *symbolic links*. A symbolic link (or "symlink") is a file that "points to" some other *entry* (another file, directory, or symbolic link). The pointed-to entry can live anywhere—in some other directory, or even in another file system. Because of symbolic links, the graph of the hierarchy isn't acyclic: As you follow a path through the hierarchy, it's possible to get into a loop. (If you leave symbolic links out of the picture, the graph *is* acyclic.)

Currently, the BeOS does *not* recognize *hard links* (although it will in the future). A hard link associates an entry in the hierarchy with a specific chunk of data (a *node*). The attraction of hard links is that more than one entry can be associated with the same node.

File System Layout and the Root File System

Before looking at the file system layout, let's get one thing clear. The term "file system" is a heavily loaded monicker:

- At the lowest level, it means a "file system format." Currently, the BeOS recognizes BFS, HFS (Macintosh), and ISO9660 (CD-ROM) file system formats (other file systems are being ported).
- "File system" can also mean the agent that knows how to turn bytes on a disk (or even in memory, as described in "Virtual File Systems") into the sort of hierarchy that the OS wants to see. When you pop in a floppy, for example, a file system handler must wrangle the disk format to convert its contents into a recognized hierarchy. When we say that the BeOS recognizes HFS, we're really saying that we provide an HFS file system handler. See "File System Handlers" for more information.
- Yet another meaning of "file system" is that part of the OS that manages and lays out all the file systems that are mounted. It's the *über*-system for all hierarchies.

In the BeOS, this *über*-system is part of the kernel. The first thing it does is mount the *root file system*. All other file systems are mounted within the root file system.

The structure of the root file system is a simple, two-level hierarchy:

- The root directory of the root file system is "/". This is the only directory in the entire hierarchy that doesn't have a parent.
- The nodes in the root file system are directories and symlinks *only*. In other words, the root file system contains no real files.

Directories at Root

The directories in the root file system have a prescribed purpose:

- The directories are *mount points* for other file systems. A mount point becomes the root directory for the file system that's mounted there. (More accurately, the *name* of the mount point becomes the *name* of the root directory; the mount point directory itself is "obscured" when a file system is mounted on it.)
- You can create your own directories in the root file system, but they can only be used as mount points—you can't use them to store files.

Symlinks at Root

The symlinks in the root file system are less restricted. The OS creates some number of symlinks automatically (as described below), but other than that, you're free to put whatever symlinks you want in the root file system. However, root level symlinks are

simply a convenience for developers—the user won't be able to see them since the Tracker doesn't display the root file system.

Default Root Nodes

When BeOS is launched, some number of directories and symlinks are automatically created in the root file system. The directories are:

- **For each "persistent" file system that the OS finds (hard disks, CD-ROM, floppy, etc.), a mount point directory is created and the file system is mounted. The mount point for the file system that was used to boot the OS is /boot; the other mount point names are taken from the storage devices themselves (if you name your floppy "fido", the floppy will be mounted at /fido).**
- **The OS also creates two *virtual file systems*, mounted at /dev and /pipe. We'll look at virtual file systems later. Briefly, a virtual file system is one that only exists in memory—it has no persistent storage. (The root file system itself is a virtual file system.)**

And the symlinks are:

- **A link is made from */boot_disk_name* to /boot.**
- **Links are made from /bin, /system, and /etc to similarly named subdirectories of /boot/beos.**
- **Links are made from /var to /boot/var and from /tmp to /boot/var/tmp.**

If you `cd` to / in a shell and list the files, you'll see something like this (where "MyDisk" is the name of the boot disk, and "fido" is a floppy):

```
l--------- 1 users 0 Jun 19 22:11 bin -> /boot/beos/bin
drwx------ 1 users 2048 Jun 19 21:58 boot
drwxr-x--- 1 users 0 Jun 19 22:11 dev
l--------- 1 users 0 Jun 19 22:11 etc -> /boot/beos/etc
drwxr-xr-x 1 users 2048 Jun 19 22:12 fido
l--------- 1 users 0 Jun 19 22:11 MyDisk -> /boot
drwxrwxrwx 1 users 0 Jun 19 22:11 pipe
l--------- 1 users 0 Jun 19 22:11 system -> /boot/beos/system
l--------- 1 users 0 Jun 19 22:11 tmp -> /boot/var/tmp
l--------- 1 users 0 Jun 19 22:11 var -> /boot/var
```

If you invoke `df`, you'll see the list of mounted file systems:

```
Mount Type Total Free Flags Device
---------------- -------- -------- -------- ----- -------------------------
/ rootfs 0 0 0
/dev devfs 0 0 0
/pipe pipefs 0 0 0
```

```
/boot bfs 532950 395715 70004 /dev/disk/scsi/050/0_2
/fido bfs 1440 904 70004 /dev/disk/floppy/raw
```

All file systems, both persistent and virtual, are displayed by `df`. The virtual file systems are the ones that have 0 size.

Navigating the File System

If your application needs to get to a well-defined watering hole (such as the directory where a user's preference settings are stored, or where apps are kept), use the symbolic pathname constants to the `find_directory()` function. The constants are listed in the "Global Constants and Defined Types" section of this chapter. `find_directory()` is documented at the end of BDirectory description.

Virtual File Systems

A virtual file system has no backing storage. The hierarchy that it represents is maintained in memory. When you shut down the OS, all virtual file systems disappear.

Currently, the BeOS provides three virtual file system handlers:

- **rootfs handles the root file system (described above). The root file system is mounted at "/". It's always the first file system to be mounted, and provides mount points for all other file systems.**
- **devfs is the handler for the *device file system*, mounted at /dev. This system contains entry points for access to hardware devices, including hard disks, CD-ROMs, and so on.**
- **pipefs is the handler for the *pipe file system*, mounted at /pipe. A "pipe" is a location through which the POSIX pipe mechanism reads and writes data.**

From the developer's perspective, a virtual file system isn't much different from a persistent file system: You can create a BVolume object to represent it, you can walk through it's hierarchy, look at its contents, and so on.

But virtual file systems are hidden from the Tracker user: The Tracker does not display virtual file systems, nor does it let the user form a query on a virtual file system.

File System Handlers

When the kernel attempts to mount a file system, if must first find a *file system handler* that understands the file system's format. In other words, the handler is an intermediary between the kernel and the bytes on a disk (for example).

All file system handlers are add-ons. The kernel loads the handler add-on that it needs, and talks to it through a set of well-defined functions. The advantage of this approach is that the kernel doesn't need to know about specific file systems, it only needs to know how to talk to the handlers.

Currently, you can't create your own file system handler. The handler API will be released sometime in the future.

The Be Book The Storage Kit The Storage Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Attribute C Functions

Declared in: be/kernel/fs_attr.h
Library: libroot.so
Summary: *more...*

The BeOS supports a powerful system of attributes that can be attached to files. Once these attributes are attached, you can query the file system for attributes whose values match certain specifications; this is discussed in the section on the **fs_query** functions, and in the section on the BQuery class.

Before you can perform these queries, however, you need to attach attributes to the file. There are two ways to do this. The typical mechanism for attaching attributes to files is through the BNode class, which provides a convenient interface for doing this. This section discusses the lower-level way to do this, by calling the C **fs_attr** functions directly.

This collection of functions also lets you look at the attributes attached to a file, get their names and sizes, and read their contents.

The **fs_attr** functions make use of directory pointers (DIR *). While these are true directory pointers, the data they hold aren't useful outside the **fs_attr** API; they're basically magic cookies that represent the file's attribute directory. Likewise, the dirent structures returned by some of these functions are also not useful outside this API.

An Example

Before dropping straight into the reference on the **fs_attr** API, let's have a look at a simple sample of how to use some of its features. The sample code fragment below opens the attribute directory for a file named **/boot/home/dirtylaundry** and scans through all the attributes in the file, fetching their names and values.

```
DIR *d;
dirent_t *ent;
attr_info info;
int fd;
char *buffer;

d = fs_open_attr_dir("/boot/home/dirtylaundry");
if (d) {
   while (ent = fs_read_attr_dir(d)) {
   fd = open("/boot/home/dirtylaundry", O_RDONLY);
   fs_stat_attr(fd, ent->d_name, &info);
   buffer = (char *) malloc((size_t) info.size);
   if (buffer)
      fs_read_attr(fd, ent->d_name, info.type, 0, buffer,
```

```
            info.size);
    ...
    close(fd);
    fs_close_attr_dir(d);
}
```

This snippet begins by opening the attribute directory for the file, using the **fs_open_attr_dir()** function. If this is successful, it returns a pointer to a directory that contains the list of attributes. A `while` loop is used to read into *ent* each attribute from the attribute directory by calling **fs_read_attr_dir()**. The information this call provides includes the size and type of the attribute, as well as its name.

Once we know the name of the attribute, we can obtain the type and size of the attribute by calling **fs_stat_attr()**; now we have all the information needed to get the value of the attribute. After allocating a buffer to contain the value of the attribute, we pass the attribute's name (ent->d_name), and the type and size (info.type and info.size) into the **fs_read_attr()** function. The value of the attribute is stored in the *buffer* we specify.

This sample skimps a bit on error handling; you'll do better, of course.

Attribute Functions

fs_close_attr_dir

int **fs_close_attr_dir**(DIR *_dirp_)

Closes the specified attribute directory. You should pass into this function the pointer returned when you called **fs_open_attr_dir**() or **fs_fopen_attr_dir**().

If successful, this function returns 0; otherwise it returns -1 and sets **errno** to a descriptive code.

- **B_FILE_ERROR**. Invalid directory reference specified.

fs_fopen_attr_dir() see fs_open_attr_dir()

fs_open_attr_dir() , fs_fopen_attr_dir()

DIR ***fs_open_attr_dir(**const char **_path_**)**
DIR ***fs_fopen_attr_dir**(int _fd_)

Opens the attribute directory for the file specified by pathname or file descriptor.

The attribute directory for a file contains a list of the attributes that are attached to the file. Once the attribute directory is open, you can use the **fs_read_attr_dir()** function to find out which attributes are present.

If the directory is opened successfully, a pointer to the directory structure is returned. This pointer should be passed to the other **fs_attr** functions to read entries from the attribute directory, as well as to close the directory when you're finished with it.

If an error occurs while opening the attribute directory, this function returns **NULL** and sets **errno** to one of the values listed below.

- **B_FILE_ERROR**. Invalid file descriptor, or a file error prevented the operation.
- **B_ENTRY_NOT_FOUND**. No matching attribute was found for the specified file.
- **B_NAME_TOO_LONG**. *path* is longer than **B_PATH_NAME_LENGTH** characters.
- **B_LINK_LIMIT**. A loop was detected in the directory structure.
- **B_NO_MEMORY**. Insufficient memory to complete the operation.
- **B_BUSY**. The specified file is currently in use.

- **B_NO_MORE_FDS**. Too many open files; all file descriptors are in use.

---

fs_read_attr

```
ssize_t fs_read_attr(int fd,
    const char *attribute,
    uint32 type,
    off_t pos,
    void *buffer,
    size_t count)
```

Reads the attribute of the type and name specified by *type* and *attribute* from the file descriptor *fd*. The attribute's data is read starting at the offset *pos* and stored in the *buffer*. *count* specifies the maximum number of bytes to be read.

The *type* can be any of the standard BeOS type definitions, such as **B_STRING_TYPE**. See the header file **be/support/TypeConstants.h** for a complete list of these types. Note that the type of the attribute is a hint only; there's no reason you can't read a **B_STRING_TYPE** attribute as an integer (except that the data would probably not make any sense if you did).

If the attribute is read successfully, `fs_read_attr()` returns the actual number of bytes of data read from the attribute into the buffer. If an error occurs, this function returns -1 sets **errno** to an appropriate value.

- **B_FILE_ERROR**. Invalid file descriptor *fd* specified.
- **B_ENTRY_NOT_FOUND**. No matching attribute was found for the specified file descriptor.

---

fs_read_attr_dir

```
struct dirent *fs_read_attr_dir(DIR *dirp)
```

Reads the current attribute from the specified attribute directory, and bumps the *dirp* so it points to the next attribute. The *dirp* pointer that you pass to this function should have been retrieved through a previous call to **fs_open_attr_dir()** or **fs_fopen_attr_dir()**.

A pointer to a **dirent** structure is returned. This structure contains information about the attribute. Do not dispose of or alter the data contained by this pointer; it belongs to the operating system.

Once a file's attribute directory has been opened, you can loop over this function to iteratively retrieve the names of all the attributes in the file. An attribute's name is recorded in the `d_name` field of the **dirent** structure; see the example in "An Example" on page193.

This function *doesn't* let you get the type or value of an attribute. For that, use **fs_stat_attr()**.

If you're looking for files that have a particular value for one or more attributes, however, you should try using the **fs_query** functions, which allow you to more easily establish complex

search operations on the values of one or more attributes.

If an error occurs (including reaching the end of the directory), this function returns **NULL**.

- **B_FILE_ERROR**. Invalid directory reference specified.
- **B_NOT_A_DIRECTORY**. The attribute directory is invalid.
- **B_ENTRY_NOT_FOUND**. You've reached the end of the attribute directory.

---

fs_remove_attr()

int **fs_remove_attr(**int *fd*, const char **attribute***)**

Deletes the named *attribute* from the file given by the file descriptor *fd*.

If the function is successful, `fs_remove_attr()` returns 0. Otherwise, it returns -1 and sets **errno** to an appropriate value.

- **B_FILE_ERROR**. *fd* is invalid, or file opened with read-only access.
- **B_BAD_VALUE**. Invalid attribute name specified.
- **B_NOT_ALLOWED**. Disk containing the file is read-only.
- **B_ENTRY_NOT_FOUND**. No matching attribute was found.

---

fs_rewind_attr_dir()

int **fs_rewind_attr_dir**(DIR **dirp*)

Rewinds the attribute directory to the beginning. This lets you start over again at the top of a file's attribute directory and read your way down toward the bottom.

Returns a result code specifying whether or not the operation was successful.

Unlike most of the other file system C functions, **fs_rewind_attr_dir()** doesn't set **errno**.

RETURN CODES

- **B_OK**. Success.
- **B_FILE_ERROR**. Invalid directory reference specified.

---

fs_stat_attr

int **fs_stat_attr**(int *fd*,

```
    const char *name,
    struct attr_info *info)
```

Returns, in the `attr_info` structure pointed to by *info*, the type and size of the specified attribute on the file whose descriptor is specified by *fd*.

The `attr_info` structure is defined as follows:

```
typedef struct attr_info
{
   uint32      type;
   off_t      size;
}
```

type contains a code defining the format of the data contained by the attribute; standard values for this field are defined in the **be/support/TypeConstants.h** header file.

size specifies the size of the data the attribute contains.

Upon success, the function returns 0. Otherwise, it returns -1 and sets **errno** to an appropriate value.

- **B_ENTRY_NOT_FOUND**. The requested attribute was not found for the file.
- **B_FILE_ERROR**. Invalid file descriptor specified.

---

fs_write_attr()

```
ssize_t fs_write_attr(int fd,
    const char *attribute,
    uint32 type,
    off_t pos,
    const void *buffer,
    size_t count)
```

Sets the value of the named *attribute* to the data in the specified *buffer*. The data's type is specified by *type* and should be one of the codes defined in **be/support/TypeConstants.h**. The length of the data to be written is specified by *count*.

At this time, writing at an offset within an attribute is not fully supported, so you should always specify a *pos* of 0 to write at the beginning of the attribute.

If the attribute already has a value, this function wholly replaces that value with the new data —even if the new data is smaller than the existing data.

Under the Be File System, files have a special attribute storage area which contains the smaller attributes. When this space is filled up, or an attribute too large to fit into the space is added to

the file, additional disk blocks are allocated for the new attributes. However, only one attribute is written per block in these additional blocks. The moral of the story is: use attributes wisely. They can be your friends, but if you overuse them, you'll bloat your files.

Upon success, the function returns the number of bytes that were actually written. Otherwise, it returns -1 and sets **errno** to an appropriate value.

- `B_BAD_VALUE`. The attribute name is too long or is an empty string.
- `B_NO_MEMORY`. Insufficient memory to complete the operation.
- `B_FILE_ERROR`. Invalid file descriptor *fd* specified, or the the file is read-only.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Attribute C Functions Function Summary

Function Summary

Function	Description
fs_fopen_attr_dir()	Opens the attribute directory for the file given by a file descriptor
fs_open_attr_dir()	Opens the attribute directory for the file given by a pathname
fs_remove_attr()	Deletes the named attribute
fs_rewind_attr_dir()	Rewinds the attribute directory to the beginning
fs_write_attr()	Writes an attribute to a file

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Query C Functions

Declared in: be/kernel/fs_query.h
Library: libroot.so
Summary: *more...*

Normally, when you want to perform a file system query, you would use the BQuery class, which is a nice, clean, object-oriented way to do it. If you have an aversion to object-oriented programming, however, or you're writing a simple C program and would rather use C functions instead, then you've come to the right place.

Be aware that, currently, you can't perform "live" queries at the C level. If you want a live query, you have to use a BQuery object.

Opening, Reading, and Closing a Query

To begin a query, you call **`fs_open_query()`**. **`fs_open_query()`** performs a "one-shot" query: It lets you ask for all the files that fulfill certain criteria right now. This criteria is expressed as a "predicate" string. This is a formula that lists the attribute values that you're interested in. A simple predicate takes the form

- *attribute op value*

For example, you can look for files that are larger than 5K by supplying a string that looks like this:

```
"size > 5000"
```

Simple predicates can be grouped and combined to create more complex predicates. Here we look for files named "fido" that are greater than 5K:

```
"(name = fido) && (size > 5000)"
```

For a full description of the predicate format, see "The Predicate, Attributes, and Indices" in the BQuery class description, but be aware of this difference:

- The `fs_query` functions don't let you "push" elements onto the predicate; all predicates must be expressed as strings.

Once the query has been opened, you can step through the files that fulfill the predicate through iterated calls to **`fs_read_query()`**. When all the winning files have been visited, **`fs_read_query()`** returns **`NULL`**.

When you've finished using your query, you must close it by using the **`fs_close_query()`** function.

An Example

The following sample demonstrates very briefly how to perform a simple, non-live query. In this example, we're searching for all C header files on the device specified by *devnum*.

```
void sample_query(dev_t devnum) {
   DIR      *q;
   dirent_t   *   d;

   q = fs_open_query(devnum, "name=*.h", 0);
   if (q) {
      while (d = fs_read_query(q)) {
      ...
      }
      fs_close_query(q);
   }
}
```

The code opens the query by calling **fs_open_query()**, then, calls **fs_read_query()** in a loop until **NULL** is returned. Once that happens, **fs_close_query()** is used to close the query.

Query Functions

fs_close_query()

```
int fs_close_query(DIR *dir)
```

Closes a query which was previously opened using the `fs_open_query()` function. You pass in the DIR * returned by either of these functions. The pointer *dir* is freed by this function.

If the query closes successfully, `fs_close_query()` returns 0; otherwise, it returns -1 and sets **errno** to an appropriate value.

- `B_FILE_ERROR`. A file system error prevented the operation from succeeding.

fs_open_query()

```
DIR  *fs_open_query(dev_t device,
    const char *query,
    uint32 flags)
```

Opens a new query on the specified device. The *query* string is the criteria or "predicate" that describes the files that you're looking for. For information on how to construct the *query* string, see the BQuery class. Note that you can't use the "push" method: fs_open_query() only understands predicate strings.

flags is currently unused; pass 0 as its value.

The pointer returned by this function is used to identify your query to the other query functions; you should not dispose of it yourself—this will be done for you when you call `fs_close_query()`. If the query couldn't be opened, the function returns **NULL** and sets **errno** to an appropriate value.

- **B_ENTRY_NOT_FOUND.**,*path* does not exist, or is **NULL** or an empty string.
- `B_BAD_VALUE`. *device* does not specify a valid device, *query* is **NULL** or an invalid expression, or a live query was requested without specifying a valid *port*.
- `B_NO_MEMORY`. Insufficient memory to complete the operation.
- `B_FILE_ERROR`. A file system error prevented the operation from succeeding.
- `B_BUSY`. A busy node could not be accessed.
- `B_NO_MORE_FDS`. All file descriptors are in use (too many open files).
- **E2BIG**. *query* expression is too big.

fs_read_query()

dirent ***fs_read_query**(DIR *d)*

Returns the next item that matches the specified query. The *d* argument should have been gotten from a previous call to `fs_open_query()`; it identifies the particular query from which to read.

You mustn't free the pointer returned to you by this function.

If an error occurs while reading the query, this function returns **`NULL`** and sets **`errno`** to an appropriate value.

- `B_FILE_ERROR`. A file system error prevented the operation from succeeding.
- `B_BAD_VALUE`. An error occurred processing the query predicate.
- `B_ENTRY_NOT_FOUND`. No more matches.
- `B_NOT_A_DIRECTORY`. A non-directory node was found where a directory was expected.
- `B_INTERRUPTED`. A signal interrupted the read.
- **`E2BIG`**. Query predicate is too big.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Query C Functions Function Summary

Function Summary

Function	Description
fs_close_query()	Close a query
fs_open_query()	Open a new query
fs_read_query()	Returns the next match for the query

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Entries and Nodes

The most important concept that you should keep in mind when you're using the Storage Kit is that a file is considered both an *entry* and a *node*:

- The entry part of a file is its location in the file hierarchy. An entry is similar to a pathname: It tells you where a file is (or should be), but it doesn't let you look at its contents.

- The node part of a file is its data. A node is an actual "thing" that's separate from the file's entry—when you rename a file, for example, all you're doing is tagging the node with a different pathname (or, in our lingo, you move the node from one entry to another). Just as entries don't know about data, nodes don't know anything about entries: A node doesn't know where its entry is located.

This concept really isn't new: If you're familiar with POSIX, then you've already dealt with entries and nodes, except you called them pathnames and file descriptors.

Entries

In the Storage Kit, entries are represented three ways:

- As pathnames,
- as **entry_ref** structures,
- and as BEntry objects.

Any entry can be given by any of these representations. Furthermore, the representations are fairly easily converted: Given an **entry_ref**, it's trivial to get a BEntry, from which you can easily get a pathname, which can be turned into an **entry_ref**. Which representation you use depends on what you're doing:

- You use pathnames or **entry_ref**s to keep track of the entries you're interested in.
- You use BEntry objects to query and manipulate the entries. For example, if you want to know if an entry is a directory or a file, you need a BEntry object.

An entry_ref or BEntry isn't necessarily valid across reboots; don't archive these and expect them to work when the system is restarted. Instead, you should save pathnames (either complete or partial, depending on your specific needs; when possible, you should use the directory constants used by the **find_directory()** function).

Nodes

Nodes are represented in two ways:

- As **`node_ref`** structures,
- and as BNode objects.

Here, again, the representations are easily converted. As for use...

- **`node_ref`**s are used for purposes that we're going to ignore for now (we're just covering the basics, here).
- The BNode class is where the action is. If you want to read and write the data in a file, you need a BNode object—more specifically, you need an instance of the BFile class, which derives from BNode.

Every node has a type, or *flavor*. There are three node flavors:

- plain files
- directories
- symbolic links

These flavors are represented by subclasses of BNode: BFile, BDirectory, and BSymLink. Note that a **`node_ref`** doesn't know its node's flavor.

More Facts

Some more facts you should be aware of:

- Every node has an entry; not every entry has a node.

If you've got your hands on a node, then you can assume that there's an entry somewhere that "contains" that node. (This isn't entirely true, but it's true enough for now. For the real story, see "Lies")

The converse isn't true: An entry needn't have any data. Such entry's are called "abstract". Abstract entries are useful for expressing the location of a file before it's created (for example). But don't be misled: Abstract entries do *not* exist in the file hierarchy, they're simply placeholders that your app uses to designate a location. This leads us to our next fact:

- Every file in the file hierarchy has an entry and a node.

This might seem obvious; if it does, then go to the next fact. For the skeptics, here's the gospel: The files that "normal" apps work with are real—they actually exist as bytes on a disk. Such files have a location in the hierarchy, and they contain data.

- You can convert an entry into a node, but not the other way around.

The BNode class accepts any form of entry representation as an argument to its constructor. In other words, given a

pathname, **`entry_ref`**, or BEntry object, you can create a BNode. But once you've got your BNode, you can't go back: There's no way to get an entry from a node.

Returning to the BNode constructor: You can *only* create a BNode by passing the constructor an entry (in one of its representations). This is an important point that we'll pick up in the next section.

Lies

Here are some more facts, slight alterations to the near truths spoken above:

A Node can Lose its Entry

Consider this scenario: You create a BFile object to some file. While you're reading and writing the file, the user deletes the file through the Tracker or from a Terminal. What the user has done is delete the node's entry, not the node itself. The node isn't destroyed until all references to the node, including your BFile, are deleted (or, more accurately, "closed"). The twist is that your BFile by itself has no way of knowing that the entry is gone.

So what are you supposed to do? In general, whenever you free a BFile object, you should first check to make sure the entry still exists; of course, the BFile itself can't tell you (remember: A node doesn't know about its entry), so you have to save the entry that was used to create the BFile. You ask the entry if it still exists, and then do whatever you have to do if it doesn't, such as alert the user, ask for a new entry name, and so on.

Unfortunately, this problem has another wrinkle: What if the user *moves* the entry while you're using the entry's node? In this case, the node isn't going to be destroyed, but if you ask the generative entry (the entry that was used to create the BFile object), it *looks* like the entry is gone.

There's no generic solution to the entire problem. Not because it's impossible to implement, but because the "right" solution depends on what the user meant by deleting or moving the entry. Most applications take this approach: The user knows files as entries, not as nodes. If a user opens a file through your app, moves the entry (through some other vehicle, such as the Tracker), and then asks your app to save the file, what the user really want is for you to save the node under the same name that was used to open the node.

A BDirectory Knows its Entry

BDirectory is an exception to the "ignorant node" rule: You can ask a BDirectory object for its entry (through its `GetEntry()` function).

The Be Book

,

...in lovely HTML...

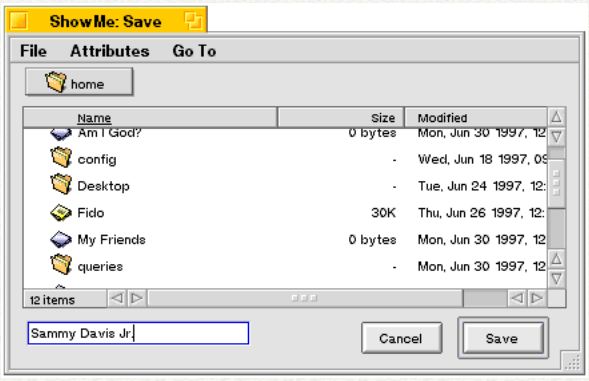
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFilePanel

Derived from: none
Declared in: be/storage/FilePanel.h
Library: libtracker.so
Summary: *more...*

BFilePanel knows how to create and display an "Open File" or "Save File" panel, and provides the means for filtering and responding to the user's actions on the panel. The Save Panel looks like this:



The Open Panel looks pretty much the same, but without the text view in the lower left corner.

Creating and Using a BFilePanel

To create and use a BFilePanel, follow these steps:

1. **Construct a BFilePanel object** in response to the user's request (most likely, a click on an "Open" or "Save"/"Save As" menu item). When you construct the panel, you have to specify its "mode" (Open or Save).

2. **Fine-tune the panel** by telling it which directory to display, whether it allows multiple selection, whether it can open a directory, which target it should send notifications to, and so on. (Most of these parameters can also be set in the constructor.)

3. **Invoke `Show()` on the panel**, and then wait for the user to confirm a selection (or close the panel).

4. **Receive a message**. When the user confirms a selection (or cancels the panel), the panel disappears and a notification message (Open, Save, or Cancel) is sent to the panel's target. The message identifies the confirmed file(s).

5. **Delete the BFilePanel object...or don't**. When the user closes a file panel, the object is not automatically deleted; you have to do it yourself. But you may not want to. If you don't delete the panel, you can simply call `Show()` the next time you want to display it; the state from

the previous invocation (the panel's size and location, the directory it points to) is remembered.

Constructing and Fine-tuning the Panel

The BFilePanel constructor has about two thousand arguments. They all have default values, and most of the parameters that they control can be set through individual functions. The following sections list and describe the constructor arguments and tell you if there's an analogous function.

Panel Mode

Argument	Default	Function
`file_panel_mode` *mode*	**`B_OPEN_PANEL`**	none

There are two file panel modes: `B_OPEN_PANEL` and `B_SAVE_PANEL`. You've got to make up your mind in the constructor.

Target

Argument	Default	Function
`BMessenger` **target*	`be_app_messenger`	`SetTarget()`

The target represents the BLooper/BHandler that will receive the Open, Save, and Cancel messages.

Panel Directory

Argument	Default	Function
`entry_ref` **panel_directory*	cwd	`SetPanelDirectory()`

When a panel is first displayed, it has to show the contents of some directory; this is called the "panel directory." The panel directory defaults to the current working directory.

Confirmable Node Flavors

Argument	Default	Function
`uint32` *node_flavors*	**`B_FILE_NODE`**	none

This parameter applies to Open panels only.

There are three node flavors: **`B_FILE_NODE`**, **`B_DIRECTORY_NODE`**, and **`B_SYMLINK_NODE`**. You combine these constants to declare the flavors that you want the user to be able to confirm. Before describing the flavor settings, keep this in mind...

- **Double-clicking a directory in the file list always enters the directory, regardless of the panel's flavor setting.**

If you understand the following, you can save yourself some reading:

- If your app wants to open files only, then stick with the default (**`B_FILE_NODE`**); the user will be able to confirm files and symlinks to files. If you want directories as well (for example, a compression app might want to work on files *and* directories) then add in **`B_DIRECTORY_NODE`** (symlinks to directories are okay, as well). If you only want directories (unusual, but possible), then leave **`B_FILE_NODE`** out of it.

If you're not convinced, read on:

- If the setting includes **`B_FILE_NODE`** and the user selects and confirms a file or a symlink to a file, the file (or symlink) is delivered to your target. If it doesn't include **`B_FILE_NODE`** and the user selects a file (or symlink to a file), the Open button is disabled.
- If the setting includes **`B_DIRECTORY_NODE`** and the user selects and Opens (i.e. clicks the Open button) a directory or a symlink to a directory, the directory (or symlink) is delivered to your target. If it doesn't include **`B_DIRECTORY_NODE`** and the user Opens a directory (or symlink to a directory), the directory is entered (the contents of the directory are displayed in the file list).
- If the setting includes **`B_SYMLINK_NODE`** and the user confirms a symlink, the symlink is delivered to your target. If it doesn't include **`B_SYMLINK_NODE`** and the user selects symlink, the panel's response depends on the inclusion of the other two flavors. Note that including **`B_SYMLINK_NODE`** is an odd thing to do—it only makes sense if it's not combined with either of the other two flavors, and even then it doesn't make much sense.

As implied by the here, when the user confirms a symlink (regardless of the flavor setting), you always receive the *symlink itself* in the Open message—you don't get the file or directory it points to.

Multiple Selection

Argument	Default	Function
`bool` *allow_multiple_selection*	**`true`**	none

This parameter determines whether the user is allowed to select more than one item at a time. Save panels should set this to **`false`**.

Notification Message

Argument	Default	Function
`BMessage` **message*	a default BMessage	**`SetMessage()`**

By default, the format of the message that's sent to your target when the user confirms or cancels is defined by the file panel (the default formats are defined later). You can override

the default by specifying your own BMessage. The BMessage is copied by the BFilePanel object. You can change this message using the `SetMessage()` function.

Ref Filter

Argument	Default	Function
`BRefFilter` **filter*	`NULL`	`SetRefFilter()`

When panel directory changes (this includes when the panel is constructed, and when the panel's `Refresh()` function is called), or when a new entry is added to the existing directory, the new entries are passed, one-by-one, to the panel's BRefFilter object through a BRefFilter hook function. In your implementation of the hook function, you can reject individual entries; rejected entries won't be displayed in the file list.

By default, a file panel has no BRefFilter. To supply one, you have to subclass BRefFilter (in order to implement the hook function) and pass it in.

- Note that the ref filter isn't asked to "re-review" the entry list when the file panel is `Show()`'d after being hidden.

Is Modal?

Argument	Default	Function
`bool` *modal*	`false`	none

A modal file panel can't be closed; to get rid of the panel, the user has to click a button. By default, file panels are not modal.

Hide When Done

Argument	Default	Function
`bool` *hide_when_done*	`true`	`SetHideWhenDone()`

By default, a file panel is hidden when the user confirms or Cancels. If you set *hide_when_done* to `false`, the panel remains on the screen. Clicking the panel's close box always hides the panel

The Target and the Messages it Sees

When the user confirms a selection or cancels a file panel, a BMessage is constructed and sent to the target of the BFilePanel object. By default, the target is `be_app_messenger`. You can specify a different target (as a BMessenger) through the BFilePanel constructor, or through the `SetTarget()` function.

The format of the BMessage that the target receives depends on whether the user is opening, saving, or canceling.

Open Notification

If the target is **be_app_messenger** and the what field is **B_REFS_RECEIVED**, the BMessage shows up in the **RefsReceived()** function. Otherwise it's sent to the target's **MessageReceived()**.

- By default, the what field is **B_REFS_RECEIVED**. You can override the default by supplying your own BMessage (**SetMessage()**).
- The "refs" field (type **B_REF_TYPE**) contains **entry_ref** structures, one for each entry that the user has confirmed.
- The message may contain other fields, as copied from the BMessage you (optionally) supplied. The data in these fields won't be changed.

Keep in mind that the refs that you receive through this message point to the literal entries that the user confirmed. In other words, if the confirmed selection is a symlink to a file, you'll receive a ref for the symlink, not the file (and similarly for a link to a directory). It's up to you to turn the symlink into a file (which is probably what you want).

If you want a BEntry object, all you have to do is pass **true** as the traverse argument to BEntry's constructor or **SetTo()**:

```
/* We'll assume that 'ref' was just plucked from an open notification. */
BEntry entry(ref, true);
```

You don't even have to check to see if the ref *is* a symlink.

If you want to turn a symlink ref into a ref to the pointed-to file, just add this line:

```
entry.GetRef(&ref);
```

Save Notification

Save notifications are always sent to the target's **MessageReceived()** function.

- By default, the what field is **B_SAVE_REQUESTED**. You can override the default by supplying your own BMessage (**SetMessage()**).
- The "directory" field (type **B_REF_TYPE**) contain a single **entry_ref** structure that points to the directory in which the user has requested the entry be saved (in other words, the ref refers to the panel directory).
- The "name" field (**B_STRING_TYPE**) is the text the user typed in the Save Panel's text view.
- The message may contain other fields, as copied from the BMessage you (optionally) supplied. The data in these fields won't be changed.

Note that if the user confirms a name that collides with an existing file, an alert is automatically displayed. The user can then back out of the confirmation and return to the Save Panel, or clobber the existing file. The save notification is sent after (and only if) the user agrees to clobber the file.

The file *isn't* clobbered by the system; it's up to you (as the receiver of the save notification) to do the dirty work.

Cancel Notification

A cancel notification is sent *whenever* the file panel is hidden. This includes the Cancel button being clicked, the panel being closed, and the panel being hidden after an open or a save (given that the panel is in hide-when-done mode).

Cancel notifications are always sent to the target's `MessageReceived()` function.

- The `what` field is always `B_CANCEL`, even if you supplied your own BMessage.
- The "old_what" field (`B_UINT32_TYPE`) records the "previous" `what` value. This is only useful (and dependable) if you supplied your own BMessage: The **`what`** from your message is moved to the "old_what" field. If you didn't supply a BMessage, you should ignore this field (it could contain garbage).
- The "source" (`B_POINTER_TYPE`) is a pointer to the BFilePanel object that was closed.
- The message may contain other fields, as copied from the BMessage you (optionally) supplied. The data in these fields won't be changed.

Keep in mind that when a file panel is closed—regardless of how it's closed—the BFilePanel object is *not* destroyed. It's merely hidden.

Modifying the Look of the File Panel

There are two ways you can modify the look of your BFilePanel object.

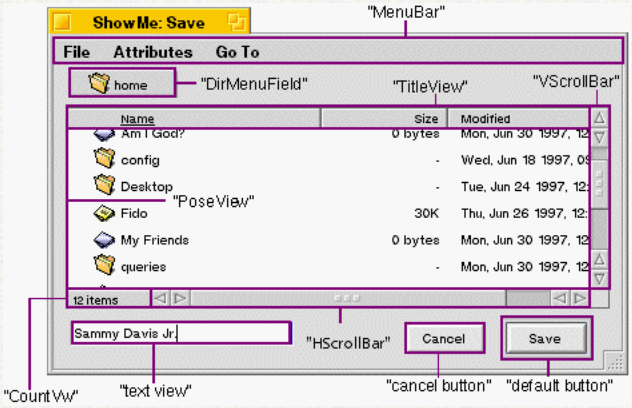
- You can do some simple text twiddling by calling the label- and text-setting functions `SetButtonLabel()` and `SetSaveText()`.
- If you need to *really* change the look, you can get a handle on the panel's BWindow and BView objects in order to move them around, add your own, or whatever. You get the window through the `Window()` function. Finding specific views within the window is described below.

Finding Views in the Panel

The views in the panel are (mostly) named, as listed and shown below

- "MenuBar" is the window's menu bar.
- "DirMenuField" is the path popup.
- "TitleView" is the bar that holds the attribute titles.
- "PoseView" is the scrollable list of files.

- "VScrollBar" and "HScrollBar" are the vertical and horizontal scroll bars.
- "CountVw" is the item counter to the left of the horizontal scroll bar.
- "text view" is where the user types a file name (Save Panel only).
- "default button" is the **Save** or **Open** button.
- "cancel button" is the **Cancel** button.



The background view doesn't have a name, but it's always the first in the window's list of views:

```
BView *background = filepanel->Window()->ChildAt(0);
```

The other views can be found by name, reckoning off of the background view. For example, here we get the "PoseView" view (the view that contains the file list):

```
BView *files = background->FindView("PoseView");
```

The C Functions

You can also display Open and Save Panels through the global C functions **`run_open_panel()`** and **`run_save_panel()`** (which are declared in **FilePanel.h**). The functions create BFilePanel objects using the default constructor settings (modulo the **`file_panel_mode`**, of course).

The C functions create a new file panel each time they're called, and delete the panel when the user is finished with it.

Hook Functions

SelectionChanged()

Invoked whenever the user changes the set of selected entries.

WasHidden()

Invoked just after the file panel is hidden because of the user's actions (it's not invoked if you call **Hide()** yourself).

Constructor and Destructor

BFilePanel()

```
BFilePanel(file_panel_mode mode = B_OPEN_PANEL,
    BMessenger* target = NULL,
    entry_ref *panel_directory = NULL,
    uint32 node_flavors = 0,
    bool allow_multiple_selection = true,
    BMessage *message = NULL,
    BRefFilter *filter = NULL,
    bool modal = false,
    bool hide_when_done = true)
```

The constructor creates a new BFilePanel object and initializes it according to the arguments. The panel isn't displayed until you invoke **Show()**. The arguments are thoroughly described in "Constructing and Fine-tuning the Panel."

You may notice that some of the default arguments shown here don't jibe with the defaults listed in the section "Constructing and Fine-tuning the Panel". In particular, the *node_flavors* argument was described as defaulting to **B_FILE_NODE**, but is shown here as 0. The "Constructing..." descriptions are correct: The default values shown here are caught and converted by the BFilePanel constructor.

~BFilePanel()

```
virtual ~BFilePanel()
```

Destroys the BFilePanel. The object's target and BRefFilter are not touched by this destruction. If the object is currently displaying a file panel, the panel is closed.

Member Functions

GetNextSelectedRef() , Rewind()

```
status_t GetNextSelectedRef(entry_ref *ref)
void Rewind(void)
```

`GetNextSelectedRef()` initializes its arguments to point to the "next" ref in the file panel's set of currently selected items. The function returns **`B_ENTRY_NOT_FOUND`** when it reaches the end of the list. `Rewind()` gets you back to the top of the list.

Although you can call these functions anytime you want, they're intended to be used in implementations of the **`SelectionChanged()`** hook function.

GetPanelDirectory() see SetPanelDirectory()

Hide() see Show()

HidesWhenDone() see SetHideWhenDone()

IsShowing() see Show()

Messenger() see SetTarget()

PanelMode()

```
file_panel_mode PanelMode(void) const
```

Returns the object's mode, either **`B_OPEN_PANEL`** or **`B_SAVE_PANEL`**. The mode is set in the constructor and can't be changed thereafter.

RefFilter() see SetRefFilter()

Refresh()

```
void Refresh(void)
```

`Refresh()` tells the file panel to re-read the contents of the panel directory, which causes the directory's entries to be re-run through the ref filter.

You don't have to call **`Refresh()`** in order to keep the panel in sync with the directory's contents —the directory and file panel are kept in sync automatically.

Rewind() see GetNextSelectedRef()

SelectionChanged()

virtual void **SelectionChanged**(void)

This hook function is invoked whenever the user changes the set of selected files. Within your implementation of this function, you iterate over **GetNextSelectedRef()** to retrieve refs to the currently selected files.

SendMessage()

virtual void **SendMessage**(const BMessenger *_messenger_, BMessage *_message_)

Sends BMessage *message* to the BHandler targeted by *messenger*.

See also: **BMessenger::SendMessage()**

SetButtonLabel() , SetSaveText()

void **SetButtonLabel**(file_panel_button *which_button*, const char **label*)
void **SetSaveText**(const char **text*)

`SetButtonLabel()` lets you set the label that's displayed in the panel's buttons. The button that a specific invocation affects depends on the value of **`which_button`**:

- **`B_DEFAULT_BUTTON`** is the Open button for an Open Panel and the Save button for a Save Panel.
- **`B_CANCEL_BUTTON`** is the "Cancel" button.

`SetSaveText()` sets the text that's displayed in the Save Panel's text view (the area in which the user types and confirms a file name).

SetHideWhenDone() , HidesWhenDone()

void **SetHideWhenDone**(bool *hide_when_done*)
bool **HidesWhenDone**(void) const

By default, a file panel is hidden when the user confirms or Cancels. You can control this behavior using the **SetHideWhenDone()** function. If you set *hide_when_done* to **`false`**, the panel remains on the screen; if you specify **`true`**, the panel hides when the user confirms or Cancels. Clicking the panel's close box always hides the panel.

HidesWhenDone() returns the current setting of this option: **true** if the panel hides when the user is done with it or **false** if it remains on the screen.

SetMessage()

void **SetMessage**(BMessage *msg)

SetMessage() allows you to set the format of the file panel's notification messages. The message can also be set through the constructor. See "The Target and the Messages it Sees" for more information.

A copy is made of the BMessage, so it is your responsibility to delete *msg* when you no longer need it.

SetPanelDirectory() , GetPanelDirectory()

void **SetPanelDirectory(**BEntry *dirEntry**)**
void **SetPanelDirectory(**BDirectory *dirObj**)**
void **SetPanelDirectory(**entry_ref *dirRef**)**
void **SetPanelDirectory(**const char *dirStr**)**

void **GetPanelDirectory(**entry_ref *ref**)** const

The **SetPanelDirectory()** function sets the panel's "panel directory." This is the directory whose contents are displayed in the panel's file list. You can also set the panel directory through the constructor. If you don't supply a directory, the current working directory is used.

GetPanelDirectory() initializes *ref* to point to the current panel directory. The argument must be allocated.

SetRefFilter() , RefFilter()

void **SetRefFilter(**BRefFilter *filter**)**

BRefFilter ***RefFilter(**void**)** const

Whenever the file panel's panel directory is changed or refreshed (**Refresh()**), or when a new entry is added to the current panel directory, the "new" entries are run through the panel's "ref filter." The BRefFilter class defines a single boolean hook function called **Filter()**. The function receives the entries, one-by-one, and can reject specific entries (because they're the wrong file type, for example). Rejected entries are not shown in the panel's file list.

The `SetRefFilter()` function sets the panel's ref filter. You can also set it through the constructor. Ownership of the filter is *not* handed to the panel. You mustn't delete the ref filter while the panel is still extant.

RefFilter() returns a pointer to the panel's ref filter.

SetSaveText() see SetButtonLabel()

SetTarget() , Messenger()

void **SetTarget(**BMessenger *bellhop***)**
BMessenger **Messenger(**void**)** const

`SetTarget()` sets the target of the file panel's notification messages. The target can also be set through the constructor. If you don't set a target, **`be_app_messenger`** is used. See the BInvoker class (in the Application Kit) for an explanation of how a BMessenger can be used as a target.

A copy is made of the BMessenger, so it is your responsibility to delete *bellhop* when you no longer need it.

`Messenger()` returns (a copy of) the messenger that's used as the file panel's target.

Show() , Hide() , IsShowing() , WasHidden()

void **Show(**void**)**
void **Hide(**void**)**
bool **IsShowing(**void**)** const
virtual void **WasHidden(**void**)**

These functions show and hide the file panel, and tell if you the panel is currently showing.

`WasHidden()` is a hook function that's invoked whenever the user's actions causes the file panel to be hidden. If you call **`Hide()`** yourself, **`WasHidden()`** is *not* invoked.

WasHidden() see Show()

Window()

BWindow ***Window(**void**)** const

Returns a pointer to the file panel's window. If you want to mess around with the window's views, see "Modifying the Look of the File Panel."

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BFilePanel Function Summary

Function Summary

Function	Description
BFilePanel()	Constructor
~BFilePanel()	Destructor
GetNextSelectedRef()	Returns the "next" ref in the set of selected items
GetPanelDirectory()	Returns the panel directory
Hide()	Hides the file panel
HidesWhenDone()	Returns hide when done flag
IsShowing()	Tests the visibility of the file panel
Messenger()	Returns a BMessenger targeting the target handler
PanelMode()	Returns the panel's mode
RefFilter()	Returns the current ref filter
Refresh()	Rescans the panel directory for changes
Rewind()	Resets the selected item pointer
SelectionChanged()	Hook called when the user changes the selection
SendMessage()	Sends a message to the target handler
SetButtonLabel()	Sets the button labels
SetHideWhenDone()	Sets hide when done flag
SetMessage()	Sets the model notification message
SetPanelDirectory()	Sets the panel directory

SetRefFilter()	Sets the ref filter
SetSaveText()	Sets the text in the text view
SetTarget()	Sets the target handler for the notification messages
Show()	Shows the file panel
WasHidden()	Hook called when a file panel is hidden by the user
Window()	Returns a pointer to the file panel's window

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRefFilter

Derived from: none
Declared in: be/storage/FilePanel.h
Library: none
Summary: *more...*

Overview

The BRefFilter class lets you filter the items that a file panel is about to display. This filtering is performed by the class' only function, **`Filter()`**. **`Filter()`** is a hook function; to use a BRefFilter, you have to create a derived class and implement the **`Filter()`** function.

To assign your BRefFilter object to a file panel, you invoke BFilePanel's **`SetRefFilter()`** function. (The BFilePanel constructor also lets you set the filter.) If you don't specifically assign a filter, the file panel will not have one—there is no "default" ref filter object. You maintain ownership of the BRefFilter that you assign to a file panel; the file panel doesn't delete or otherwise change your object.

You can assign the same filter to more than file panel. However, the **`Filter()`** function isn't told which panel it's being invoked for.

Hook Functions

`Filter()`
Look down there.

Member Functions

Filter()

```
virtual bool Filter(const entry_ref *ref,
    BNode *node,
    struct stat *st,
    const char *filetype)
```

`Filter()` is a hook function that's invoked whenever the file panel to which it's been assigned reads the entries in its "panel directory." The function is invoked once for each entry in the directory. All the arguments to the function refer to the entry currently under consideration. (Note that the function is never sent an abstract entry, so the *node*, *st*, and *filetype* arguments will always be valid.)

Your implementation of **`Filter()`** can use any or all of the arguments to figure out if the entry is a valid candidate for display in the file panel's file list. Simply return **`true`** or **`false`** to indicate if the entry is a winner or a loser.

Technically, **`Filter()`** is invoked when...

- ...the file panel's panel directory is set, either through the BFilePanel constructor or the **`SetPanelDirectory()`**, and when...
- ...the file panel's **`Refresh()`** function is called.

A BRefFilter can be assigned to more than one BFilePanel object (assignation is performed through BFilePanel's constructor or **`SetRefFilter()`** function). But it's not probably not a great idea to do so: At any particular invocation of **`Filter()`**, the BRefFilter doesn't which BFilePanel object it's working for.

You maintain ownership of the BRefFilter objects that you create. Assigning a ref filter to a file panel does *not* hand ownership of the BRefFilter to the BFilePanel. You shouldn't delete a BRefFilter while a BFilePanel is still using it; but it's your responsibility to delete it when it's done.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BRefFilter Function Summary

Function Summary

Function	Description
Filter()	Hook called to decide on acceptability of an entry_ref

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

File Panel Messages

The file panel produces three messages: **B_REFS_RECEIVED**, **B_SAVE_REQUESTED**, and **B_CANCEL**. The first of these was discussed in "General Messages". It's produced when the user picks files to open from the panel. The other two messages are described below.

B_SAVE_REQUESTED

The file panel produces this message when the user asks the application to save a document. It has two data fields:

Field	Type code	Description
"directory"	**B_REF_TYPE**	An **entry_ref** referring to the directory where the document should be saved.
"name"	**B_STRING_TYPE**	The file name under which the document should be saved.

B_CANCEL

A cancel notification is sent *whenever* a file panel is hidden. This includes the Cancel button being clicked, the panel being closed, and the panel being hidden after an open or a save.

Field	Type code	Description

"old_what"	**B_INT32_TYPE**	The "previous" **what** value. This is only useful (and dependable) if you supplied the BFilePanel with your own BMessage: The **what** from your message is moved to the "old_what" field. If you didn't supply a BMessage, you should ignore this field (it could contain garbage).
"source"	**B_POINTER_TYPE**	A pointer to the BFilePanel object.

See the BFilePanel class in **The Storage Kit** chapter for more information.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

File System Info C Functions

Declared in: be/kernel/fs_info.h
Library: libroot.so
Summary: *more...*

From time to time, it can be useful to know certain information about the capabilities of the file system on a device. While the BVolume class provides you easy access to this information, it can occasionally be helpful to have more direct access to this information.

This section describes three C functions which can be used to obtain information about the file system on a device. One of these functions, `fs_stat_dev()`, returns this information given a device number. The other two functions, `dev_for_path()` and `next_dev()`, provide two ways to obtain a device number for use with `fs_stat_dev()`.

Note that these functions *don't* set **errno**.

C Functions

dev_for_path()

```
dev_t dev_for_path(const char *path)
```

Given a pathname, returns the device number of the device on which the path is located. If the result is negative, it is a return code specifying an error.

RETURN CODES

- **B_ENTRY_NOT_FOUND.**,*path* does not exist, or is **NULL** or an empty string.
- B_BAD_VALUE. *path* is null or an empty string.
- B_NAME_TOO_LONG. *path* is too long.
- B_NO_MEMORY. Insufficient memory to complete the operation.
- B_FILE_ERROR. A file system error prevented the operation from succeeding.

next_dev()

```
dev_t next_dev(int32 *pos)
```

The `next_dev()` function allows you to iterate through all devices, receiving their device numbers as a result each time. If the result is negative, it is an error code. When the end of the device list is reached, the return value B_BAD_VALUE is returned.

You should initially set *pos* to 0, then call `next_dev()` in a loop to obtain each device number until an error occurs. For example:

```
void ScanDevices(void) {
   int pos;

   pos = 0;
   while(next_dev(&pos) >=0) {
      do_something(pos);
   }
}
```

RETURN CODES

- B_BAD_VALUE. No matching device found.

fs_stat_dev()

```
int fs_stat_dev(dev_t dev, fs_info *info)
struct {} fs_info
```

`fs_stat_dev()` returns information about the specified device. This can be used in conjunction with `next_dev()` to scan all devices and record information your application requires.

This function returns 0 if the request was successful or -1 if an error occurred. Use the **`errno()`** function to determine what error in particular occurred.

The `fs_info` structure is defined as:

```
typedef struct fs_info {
        dev_t     dev;
        ino_t     root;
        uint32    flags;
        off_t     block_size;
        off_t     io_size;
        off_t     total_blocks;
        off_t     free_blocks;
        off_t     total_nodes;
        off_t     free_nodes;
        char      device_name[128];
        char      volume_name[B_FILE_NAME_LENGTH];
        char      fsh_name[B_OS_NAME_LENGTH];
    };
```

The structure's fields are:

- `dev`. The device number of the device.
- root. The inode of the root of the device.
- **`flags`**. Flags describing the device's capabilities.
- `block_size`. The fundamental block size of the device.
- `io_size`. Optimal I/O size of the device.
- `total_blocks`. The total number of blocks on the device.
- `free_blocks`. The number of free (unused) blocks on the device.
- `total_nodes`. The total number of nodes on the device.
- `free_nodes`. The number of free (unused) nodes on the device.

- `device_name`. Name of the device holding the file system.
- `volume_name`. Name of the volume contained by the device.
- `fsh_name`. Name of the file system handler for the device.

The `flags` can be any combination of the following values, which specify the attributes of the file system on the device:

- **`B_FS_IS_READONLY`**. The file system on the device is read-only.
- **`B_FS_IS_REMOVABLE`**. The device contains removable media.
- **`B_FS_IS_PERSISTENT`**. Data written to the device remains even while the device is off.
- **`B_FS_IS_SHARED`**. The file system is being shared on a network.
- **`B_FS_HAS_MIME`**. The file system supports the MIME typing system used by the BeOS.
- **`B_FS_HAS_ATTR`**. The file system supports node attributes.
- **`B_FS_HAS_QUERY`**. THe file system supports the BeOS query mechanism.

The information in the `fs_info` structure is guaranteed to be internally consistent, but the structure as a whole should be considered to be out-of-date as soon as you receive it. It provides a picture of a device as it exists just before the info-retrieving function returns. In particular, the number of free blocks and of free nodes can easily change immediately after you receive this information.

RETURN CODES

- `B_OK`. The device was found; *info* contains valid information.
- `B_BAD_VALUE`. *dev* doesn't identify an existing device.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

File System Info C Functions

Function Summary

Function Summary

Function	Description
dev_for_path()	Returns the device on which a given path resides
fs_stat_dev()	Returns information about the specified device
next_dev()	Device iterator

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Global C Functions

The next four sections document the straight C interface to attributes, indices, queries, and file system information. Technically, these functions are part of the Kernel Kit—their definitions live in header files in be/kernel, and their code is in **libroot.so**.

These functions use a global error variable (an integer), called **errno**, to register errors. You can look at the **errno** value directly in your code after a file system function fails. Alternatively, you can use the **errno()** function which prints, to standard error, its argument followed by a system-generated string that describes the current state of **errno**.

- Each thread maintains its own **errno** variable.
- **errno** is only set if there's an error—it never indicates success.
- **errno** is never cleared. If call A fails and then you call B, C, and D, **errno** will still record the error from A.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

MIME and File Types

MIME (Multipurpose Internet Mail Extensions) is a way to describe the content or intent of a parcel of data. As the name implies, MIME was invented to make e-mail smarter: Want to send a GIF image through e-mail? If your mail program understands MIME, it can encode the image, tag it with a MIME string (in this case, "image/gif"), as well as some other header information (such as the encoding protocol) before sending the message out across the wires. Any MIME-savvy recipient program will be able to properly display the message based (primarily) on the content-type string.

MIME in the BeOS

The Be file system adds a MIME string to every file in the (native) file system, and tries to figure out an appropriate MIME type for foreign files, as well. MIME types are used by the Tracker, for example, to figure out what icon it should use to represent a particular file. Another example: When the user double-clicks a file, the MIME type (for that file) is used to identify the application that will open the file. Your application can perform similar deductions: When the user drag-drops a file, your application can look at the file's MIME type and proceed (or reject) accordingly.

There are two parts to Be's MIME support:

- **File Types**. The first part is the inclusion of a MIME string as part of every file, as mentioned above. This is

called the file's *file type*. The MIME string is added as an attribute—it's not part of the file's data. Every file—whether it's a document, an application, or even a directory has its own file type attribute. For more on getting and setting a file's type, see the BNodeInfo class.

- **The File Type Database**. The second part is the system's *File Type database*. The database contains information about the file types that the system recognizes. The user can get to the database through the **`FileTypes`** preferences application. Programmatically, you access it through the BMimeType and BAppFileInfo classes.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Storage Kit

The Storage Kit is a set of C++ classes and C functions that let you access the file system; in particular, the kit lets you...

- navigate the file hierarchy
- read and write files
- monitor changes to specific files (the Node Monitor)
- ask for a set of files based on their characteristics (queries and attributes)

The basic functionality of the kit—such as reading and writing files—is also provided by the Be-supported POSIX functions (`open()`, `read()`, `write()`, `close()`, and so on) such that if you don't want to use the Storage Kit, you don't have to. But keep in mind that the Storage Kit's classes are *not* covers for POSIX functions. The POSIX functions and the Storage Kit classes are separate branches that dissolve into the same implementation—the one isn't implemented in terms of the other. POSIX is straight C, so it avoids the C++ overhead; but language efficiency aside, the POSIX functions aren't innately faster than the Storage Kit class functions.

The Storage Kit includes some additional C functions (beyond the standard POSIX implementation) so that almost everything that can be done from the C++ level can also be done in C, and vice versa. There are a few exceptions, notably these two:

- You can't create a file system index from the C++ classes.
- You can't create a live query with the C functions.

The next few sections provide some basic Storage Kit concepts, but most of the interesting stuff is in the class descriptions. The C functions that "imitate" the C++ layer are described at the end of the chapter.

The standard POSIX functions are not documented in this book. Refer to an authorized POSIX pamphlet if you're interested in learning more about these functions.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Node Monitor Function Summary

Function Summary

Function	Description
stop_watching()	Ends change notification for a target
watch_node()	Begin notification of changes to a node or volume

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Data Containers

A few constants identify messages as data containers. The system currently uses the **B_MIME_DATA** and **B_SIMPLE_DATA** constants to mark the containers it constructs for drag-and-drop operations.

B_MIME_DATA

This message constant indicates that all the data in the message is identified by MIME type names. The type code of every data field is **B_MIME_TYPE** and the name of each field is the MIME type string.

As an example, a BTextView object puts together a **B_MIME_DATA** message for drag-and-drop operations. The message has the text itself in a field named "text/plain"; the **text_run_array** structure that describes the character formats of the text is in a field named "application/x-vnd.Be-text_run_array".

B_SIMPLE_DATA

This message is a package for a single data element. If there are multiple data fields in the message, they present the same data in various formats.

For example, when the user drags selected files and directories from a Tracker window, the Tracker packages

entry_ref references to them in a **B_SIMPLE_DATA** message. The references are in a "refs" array with a type code of **B_REF_TYPE**. In other words, the message has the same structure as a **B_REFS_RECEIVED** message, but a different `what` constant.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Reply Messages

The following three messages are sent as replies to other messages.

B_MESSAGE_NOT_UNDERSTOOD

This message doesn't contain any data. The system sends it as a reply to a message that the receiving thread's chain of BHandlers does not recognize. See **`MessageReceived()`** and **`ResolveSpecifier()`** in the BHandler class of the Application Kit.

B_NO_REPLY

This message doesn't contain any data. It's sent as a default reply to another message when the original message is about to be deleted. The default reply is sent only if a synchronous reply is expected and none has been sent. See the **`SendReply()`** function in the BMessage class of the Application Kit.

B_REPLY

This constant identifies a message as being a reply to a previous message. The data in the reply depends on the circumstances and, particularly, on the original message. For replies to scripting messages, it generally has a "result" field

with requested data and an "error" field with an error code reporting the success or failure of the scripted request.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

System Messages Introduction

This chapter lists and describes the messages that are defined by Be. This includes messages that are sent from the system to your application, messages that your application can create and send to other applications (and servers), and messages that are defined for their formats, but that you don't literally *send* anywhere (such as a clipboard message format).

Most messages are listed by their command constants (`B_ABOUT_REQUESTED`, `B_QUIT_REQUESTED`, etc.). Where the command constant isn't specific enough, the message is listed by some other field. For example, every Node Monitor message has the command constant `B_NODE_MONITOR`, and is further distinguished—and listed below—by its "opcode" field value (`B_ENTRY_REMOVED`, `B_STAT_CHANGED`, etc.). If a message doesn't have a defined command constant or other descriptive field, it's identified prosaically ("Clipboard Data", for example).

Reading a Message Description

Each message entry follows this format:

Purpose: Tells you whether the message is meant to be delivered ("Deliverable"), or if it's defined simply for its format ("Format"). A message can be both: It can be used as a deliverable by one part of the system, and used just for its format in another.

If the message is deliverable, it will describe the following:

Source: The sender of the message. An important point here is whether the message is sent by the system, by your app, or by both. If the "**Source:**" line doesn't mention that your app can send the message, then don't send it. Often the source is listed, vaguely, as "the system." This denotes that the message is sent by one of the basic servers (App Server or Input Server, mainly). Other sources (Node Monitor, Roster Monitor, MIME Monitor) are identified more precisely. In any case, the identity of the source is provided mainly so you know which ballpark you're in—the exact identity shouldn't matter to your code.
Target: The target of the message. This is the BHandler object that receives the message, and, if applicable, converts it to a hook function. For example, the target of a **`B_VIEW_MOVED`** message is the view's BWindow, not the BView itself (because the BView never actually sees the **`B_VIEW_MOVED`** message—it only sees the **`ViewMoved()`** hook function). The target is often either **`be_app`** or a BWindow object, but for some messages the target is declared by your app (the target of a BControl's invocation message, for example).
Hook: The hook function that the target invokes when the message is received. If this line is missing, the message doesn't map to a hook function.

This is followed by a brief description that tells you the circumstances under which the message is sent, and how your app is expected to behave when it receives the message. If the description doesn't say what your app is supposed to do, then it doesn't *have* to actively do anything, but it shouldn't interfere with the default mechanism.

After the description is a table that lists the message's Be-defined fields. If it doesn't have any fields, the table is excluded. If a field holds an array of data, the maximum size of the array, if known, is given in brackets after the field name:

```
"byte" [3]
```

In some cases, the array size is the value of some other field. Here, the size of the array in the "argv" field is given by the value of the "argc" field:

```
"argv" ["argc"]
```

,
...in lovely HTML...
for BeOS Release 5.

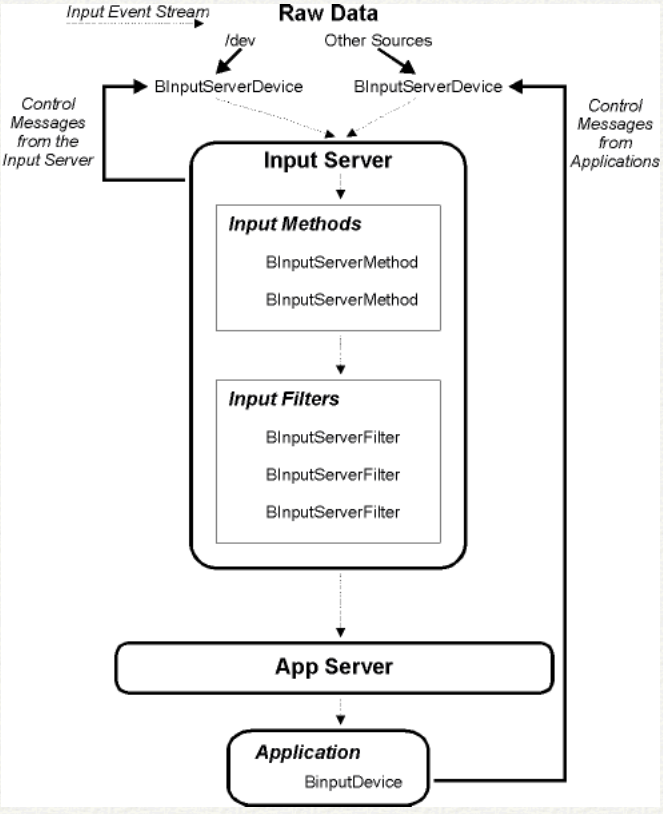
Copyright © 2000 Be, Inc. All rights reserved..

The Input Server

The Input Server is a system service that accepts user events (on the mouse and keyboard, typically) at one end, and dispatches them to the App Server at the other. In between, the Input Server runs the events through a series of filters that can inspect and modify them. The generation and filtering of events is performed by the add-ons that the Input Server loads; the Server itself just provides the plumbing. Event-generating add-ons (called *input devices*) typically correspond to one or more device drivers, although this isn't a requirement. An event-filtering add-on (*input filter*) processes the events that are fed to it; input filters aren't intended to correspond to hardware. A third type of Input Server add-on—an *input method*—is used to implement input method mechanisms, which convert keyboard events into character sets that can't be easily represented on a standard keyboard, such as Kanji.

Each of these add-on types (input devices, filters, and methods) is represented by a C++ class: BInputServerDevice, BInputServerFilter, and BInputServerMethod. For each add-on you want to create, you subclass the appropriate class and override its hook functions. An additional class—BInputDevice—lets a "normal" application send messages back through the Input Server to the input devices; a BInputDevice object can be useful if you're creating a preference app for a custom Input Server add-on, for example.

A map of the Input Server world looks like this:



Note that the Input Server and its add-ons (and BInputDevice) all live in user space, so, in theory, there's nothing that a "normal" application can do that an Input Server add-on can't do. However, Input Server add-ons are loaded early in the boot process, before some system services (such as the Media and Network servers) have started. Attempting to use services from these servers before they've started is a good way to wedge the system.

The BeOS provides a few Input Server add-ons: It installs input devices that handle a variety of mice and keyboard drivers, and an input filter that the Screen Saver engine uses to detect user activity (on the mouse and keyboard). BeOS's only built-in input method is installed when you choose the Japanese language option during the installation process.

Currently, events that are generated by the BeOS joystick drivers do not go through the Input Server.

Drivers and Input Devices

As mentioned above, most input devices (i.e. input-generating add-ons) correspond to one or more device drivers. For example, the BeOS **mouse** input device manages all the mouse drivers that the OS provides.

It's important to keep in mind that an input device is not the same as the device driver(s) it manages—they're separate pieces of code that execute in separate address spaces: the drivers run in the kernel, the add-ons run in the Input Server. An input device can **`open()`** a driver, but it must not explicitly load the driver. In other words, the add-on shouldn't re-invent or subvert the kernel's driver-loading mechanism.

Similar to drivers, Input Server add-ons must be scrupulous about managing their memory and threads:

- Memory that an add-on allocates must be freed when the add-on is unloaded, otherwise the add-on will leak.
- The hook functions that are invoked on your add-on are executed in threads that must stay as "live" as possible. If your add-on does a lot of processing that can be performed asynchronously—for example, if it's an input device that's "watching" a piece of hardware—the add-on should spawn a thread.

Building

Like all add-ons, Input Server add-ons are compiled as shared libraries. The add-ons must link against **input_server**, renamed (as a symbolic link) to **_APP_**. In other words, you set up a symbolic link like this:

```
$ cd <yourProjectDirectory>
$ ln -s /boot/beos/system/servers/input_server _APP_
```

And then link against **_APP_**.

Installing

The input server looks for add-ons in the "**input_server**" directory within **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**. Where you install your add-ons depends on what type of add-on it is:

- **input_server/devices** is for input devices
- **input_server/filters** is for input filters
- **input_server/methods** is for input methods
- You can install your input devices in the latter two directories—i.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.

- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add-ons that are supplied by the BeOS.

Loading

The Input Server automatically loads (or attempts to load) all add-ons at boot time.

Currently, the Input Server doesn't dynamically load add-ons. This is a particular annoyance if you're developing and testing an add-on. To work around this lack, move your add-on into the appropriate directory, and then quit and restart the Input Server from a Terminal:

```
/system/servers/input_server -q
```

This will gracefully shutdown the Input Server and then re-launch it. The first thing the Server does when it comes back up is re-load the add-ons from its add-on directories.

Your mouse and keyboard (and other input devices) will go dead for a moment while this is happening. This is normal.

Input Server and You

The Input Server gives applications a chance to take advantage of useful features present in input devices more interesting than your typical 101-key keyboard and 3-button mouse.

Mice and Tablets

The Input Server extends the plain **B_MOUSE_MOVED** message (which triggers a BView's **MouseMoved()** function) beyond its ordinary existence to let things like tablets pass along extra information about a user's actions. For example, drawing tablets can track the user's movement with greater precision than a mouse, and can include drawing pressure and tilt information. Some also include an "eraser."

If an application can do something useful with this information (and let's face it; drawing applications that respond to pressure and tilt on a drawing pad are useful as well as being cool), it'll be present in the **B_MOUSE_MOVED** message:

```
void MyView::MouseMoved(BPoint *where, uint32 transit, BMessage *drag_msg)
{
   BMessage *moved_msg = Window()->CurrentMessage();
   ...
}
```

The extra information that a "mouse" input device could add to the **B_MOUSE_MOVED** messages includes:

- more precise position information
- drawing pressure
- pen tilt
- "eraser" mode

Precision Position Information

Tablets store the absolute position of the pointer with as much precision as they can in the *be:tablet_x* and *be:tablet_y* fields:

```
float x, y;
x = moved_msg-FindFloat( "be:tablet_x" );
y = moved_msg->FindFloat( "be:tablet_y" );
```

These entries will be scaled to the range [0.0 to 1.0].

Pressure

Tablet pressure is stored as a float in the range [0.0 to 1.0] (minimum to maximum), present in the *be:tablet_pressure* field:

```
float pressure;
pressure = moved_msg->FindFloat( "be:tablet_pressure" );
```

Tilt

Pen tilt is expressed as a pair of floats in the range [0.0 to 1.0], where (-1.0, -1.0) tilts to the left-top, (1.0, 1.0) tilts to the right-bottom, and (0.0, 0.0) is no tilt. These floats are found in the *be:tablet_tilt_x* and *be:tablet_tilt_y* fields:

```
float tilt_x, tilt_y;
tilt_x = moved_msg->FindFloat( "be:tablet_tilt_x" );
tilt_y = moved_msg->FindFloat( "be:tablet_tilt_y" );
```

Eraser Mode

The pen's eraser mode is expressed as an int32 in the *be:tablet_eraser* field:

```
int32 erase_mode;
erase_mode = moved_msg->FindInt32( "be:tablet_eraser" );
```

A value of 1 means the pen is reversed (i.e. the eraser is on) and 0 means the pen is behaving normally. Other eraser modes may be defined in the future.

Supporting Input Methods in Views

When the user is entering text using an input method, such as the Japanese language input method that became an installation option in R4, there are two ways that applications can handle their input:

- *in-line*: the text entry interface object lets them enter text directly
- *bottom-line*: the input method itself pops up a window to accept the user's input, and then passes `B_KEY_DOWN` messages simulating the characters to the application; the app doesn't have to do anything to support bottom-line input

If your application's text-entry needs are met by the Interface Kit's BTextControl and BTextView objects, it'll automatically use the in-line mode, which gives the user a much better experience. If you're writing your own text widget, you'll have to do a little work to let the user input text directly.

Doing this is a very good idea; making your application behave well when dealing with foreign (to you) languages will improve your application's acceptance around the world.

Messages from Input Methods

When interacting with an input method, your view's `MessageReceived()` function will receive `B_INPUT_METHOD_EVENT` messages; inside is a *be:opcode* field (an int32 value) indicating the kind of event:

- `B_INPUT_METHOD_STARTED`
- `B_INPUT_METHOD_STOPPED`
- `B_INPUT_METHOD_CHANGED`
- `B_INPUT_METHOD_LOCATION_REQUEST`

`B_INPUT_METHOD_STARTED` tells your view that a new input transaction has begun. Inside the message is a BMessenger named *be:reply_to*; you should store this because it's your only way of talking to the input method while the transaction is going on.

`B_INPUT_METHOD_STOPPED` lets you know the transaction is over; you should discard the BMessenger at this point because it's gone stale.

In between the `B_INPUT_METHOD_STARTED` and `B_INPUT_METHOD_STOPPED` messages, you'll receive various `B_INPUT_METHOD_CHANGED` and `B_INPUT_METHOD_LOCATION_REQUEST` messages as the transaction proceeds.

`B_INPUT_METHOD_CHANGED` does most of the work in an input transaction; its message contains the following important fields:

Entry	Type	Description
be:string	`B_STRING_TYPE`	The text the user is currently entering; display it at the insertion

		point. BTextView also highlights the text in blue to show that it's part of a transitory transaction.
be:selection	B_INT32_TYPE	A pair of B_INT32_TYPE offsets into the *be:string* if any of the text *be:string* is currently selected by the user. BTextView highlights this selection in red instead of drawing it in blue.
be:clause_start	B_INT32_TYPE	Zero or more offsets into the *be:string* for handling languages (such as Japanese) that separate a sentence or phrase into numerous clauses. An equal number of *be:clause_start* and *be:clause_end* pairs delimit these clauses; BTextView separates the blue/red highlighting wherever there is a clause boundary.
be:clause_end	B_INT32_TYPE	Zero or more offsets into *be:string*; there will be as many *be:clause_end* entries as there are *be:clause_start*.
be:confirmed	B_BOOL_TYPE	True when the user has entered and "confirmed" the current string and wishes to end the transaction. BTextView unhighlights the blue/red text and waits for a B_INPUT_METHOD_STOPPED (to close the transaction) or another B_INPUT_METHOD_CHANGED (to start a new transaction immediately).

B_INPUT_METHOD_LOCATION_REQUEST is the input method's way of asking you for the on-screen location of each character in your representation of the *be:string*. This information can be used by the input method to pop up additional windows giving the user an opportunity to select characters from a list or anything else that makes sense. When you get a B_INPUT_METHOD_LOCATION_REQUEST, reply to the *be:reply_to* messenger (that you saved from the B_INPUT_METHOD_STARTED message) with a B_INPUT_METHOD_EVENT message, filling in the following fields:

Entry	Type	Description
be:opcode	B_INT32_TYPE	Use B_INPUT_METHOD_LOCATION_REQUEST for the opcode.
be:location_reply	B_POINT_TYPE	The co-ordinates of each character (there should be one *be:location_reply* for every character in *be:string*) in screen co-ordinates not view or window co-ordinates).
be:height_reply	B_FLOAT_TYPE	The height of each character in *be:string*.

App and Input Events

If you're writing an application and want to record or react to input events without writing an Input Server add-on (which, of course, requires an Input Server restart), you can:

1. Create a window off-screen, at a co-ordinate like (-10.0, -10.0).

2. Add a view to the window at (0.0, 0.0).

3. `Show()` and then `Hide()` the window; this is necessary or the App Server won't send you any messages.

4. Move the hidden window to (0.0, 0.0).

5. Implement the window's `DispatchMessage()` function to handle `B_KEY_DOWN`, `B_MOUSE_UP`, or whatever other input events you're interested in observing.

Modifying these messages won't affect any other applications in the system; by the time they reach your application, they've already passed through the Input Server.

You can see this trick in action in Doug Fulton's masterful Whistle application (found at **ftp://ftp.be.com/pub/samples/midi_kit/Whistle.zip**).

The Be Book The Input Server The Input Server Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Input Functions Function Summary

Function Summary

Function	Description
get_click_speed()	Returns the double click delay
get_key_info()	Returns information about the keyboard state
get_key_map()	Returns the system key map
get_key_repeat_delay()	Returns the keyboard repeat delay
get_key_repeat_rate()	Returns the keyboard repeat rate
get_keyboard_id()	Returns the keyboard identifier
get_mouse_map()	Returns the mouse button mapping
get_mouse_speed()	Returns the mouse acceleration speed
get_mouse_type()	Returns the number of recognized mouse buttons
modifiers()	Returns the current keyboard modifier bits
set_click_speed()	Sets the double click delay
set_key_repeat_delay()	Sets the keyboard repeat delay
set_key_repeat_rate()	Sets the keyboard repeat rate
set_keyboard_locks()	Sets the keyboard locks
set_modifier_key()	Maps a modifier to a particular key
set_mouse_map()	Sets the mouse button mapping
set_mouse_speed()	Sets the mouse acceleration speed
set_mouse_type()	Sets the number of recognized mouse buttons

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDeskbar

Derived from: (none)
Declared in: be/be_apps/Deskbar/Deskbar.h
Library: libbe.so
Allocation: Constructor or on the stack
Summary: *more...*

BDeskbar lets you query for the **Deskbar**'s location, orientation (expanded or contracted), and frame, and lets you add and remove items from the shelf. The class lets you expand/contract the **Deskbar** and move it to one of the pre-defined locations, but it doesn't let you remove or hide the **Deskbar**, nor does it let you replace it with a UI object of your own.

You don't need a **be_app** object to create and use a BDeskbar.

Constructor and Destructor

BDeskbar() , ~Deskbar()

BDeskbar(void)
~BDeskbar()

The constructor creates and returns a new BDeskbar object. The destructor destroys it.

Member Functions

AddItem() , RemoveItem() , CountItems() , HasItem() , GetItemInfo()

status_t **AddItem(**BView **view*, int32 **id*=NULL**)** status_t **AddItem(**entry_ref **addon*, int32 **id*=NULL**)**
status_t **RemoveItem(**int32 *id***)** status_t **RemoveItem(**const char **name***)**
int32 **CountItems(**void**)** const
bool **HasItem(**int32 *id***)** const bool **HasItem(**const char **name***)** const
status_t **GetItemInfo(**int32 *for_id*, const char ***found_name***)** status_t **GetItemInfo(**const char **for_name*, int32 **found_id***)**

A view that's sitting on the **Deskbar**'s shelf may not call any of these functions.

The **Deskbar** contains a shelf that contains replicants (archivable BViews). Typically, these replicants monitor or control some service. For example, the BeOS provides shelf items that monitor and control the input method mechanism, PPP, and the mail daemon (the date/time view is *not* a shelf replicant):

AddItem() puts a new item on the **Deskbar**'s shelf. *view*, the BView that will be displayed on the shelf, must be archivable (see BArchivable). An item on the shelf is identified by name and an integer id. The name is that of the view itself (i.e., as assigned in the BView constructor); the id is generated by the **Deskbar** and is guaranteed to be unique. *id*, if supplied, is set to the added item's unique id number.

You can also add an item to the **Deskbar** by passing an entry_ref, *addon*, to the **Deskbar** add-on to place there.

RemoveItem() removes the shelf item identified by *name* or *id*.

CountItems() returns the number of items currently on the shelf (keep in mind that it doesn't count the date/time view).

HasItem() returns true if the Deskbar shelf contains the item identified by *name* or *id*.

GetItemInfo() points **found_name* to the name of the item identified by *for_id*, or sets *found_id* of the item identified by *for_name*.

The caller is responsible for freeing *found_name*.

RETURN CODES

- `B_OK`. The request to add, remove, or get info on the item was successfully communicated to the **Deskbar**. Note that this doesn't mean that the function actually did what it was supposed to do.
- `B_BAD_VALUE`. (`GetItemInfo()`) **found_name* is `NULL`.
- Negative values. A message-sending error occurred.

CountItems() see AddItem()

Expand() see Location()

Frame()

BRect **Frame**(void) const

Returns the **Deskbar**'s frame in screen coordinates.

GetItemInfo() see AddItem()

HasItem() see AddItem()

IsExpanded() see Location()

Location() , IsExpanded() , SetLocation() , Expand()

deskbar_location Location(bool **isExpanded*=NULL) const
bool **IsExpanded**(void) const
status_t **SetLocation**(**deskbar_location** *location*, bool *expanded*=false)
status_t **Expand**(bool *expand*)

`Location()` returns a symbolic description of the **Deskbar**'s current location; see **deskbar_location** for the list of pre-defined locations. *isExpanded* (if supplied) is set to **true** if the **Deskbar** is expanded, and **false** if it's contracted; `IsExpanded()` returns the expansion value directly. Expansion and contraction is variable only if the `Deskbar`'s location is left-top or right-top; for all other locations, the expansion state is hard-wired. See **deskbar_location** for illustrations.

`SetLocation()` sets the **Deskbar**'s location *and* expands/contracts the **Deskbar**; for some locations, the expansion/contraction is hard-wired. `Expand()` expands/contracts the **Deskbar**

(if the setting isn't hard-wired) without setting its location. You should very rarely need to call these functions. Moving and expanding the **Deskbar** is in the user's domain.

RETURN CODES

`SetLocation()` and `Expand()` return...

- B_OK. The new location or expansion request was successfully communicated to the **Deskbar**. Whether the parameters were actually enforced isn't indicated.
- *Negative values*. The **Deskbar** isn't running, or some other message-sending error occurred.

- RemoveItem() see AddItem()
- Set see Location()

Deskbar Constants

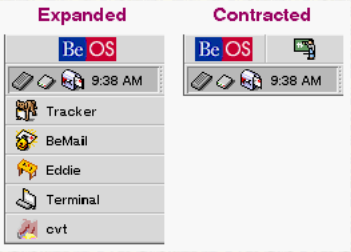
Deskbar Location

enum **deskbar_location** Constant	Meaning
B_DESKBAR_TOP	Expanded (only) along the top.
B_DESKBAR_BOTTOM	Expanded (only) along the bottom.
B_DESKBAR_LEFT_BOTTOM	Contracted (only) in bottom left corner.
B_DESKBAR_RIGHT_BOTTOM	Contracted (only) in bottom right corner.
B_DESKBAR_LEFT_TOP	In the top left corner (expanded or contracted).
B_DESKBAR_RIGHT_TOP	In the top right corner (expanded or contracted).

The **deskbar_location** constants are used to set and return the Deskbar's location (see Location()). The six locations are shown in the two illustrations below:

The **deskbar_location** value affects the Deskbar's expanded state: The Deskbar can be expanded *or* contracted in B_DESKBAR_LEFT_TOP and B_DESKBAR_RIGHT_TOP locations only. In the

other locations, the expansion/contraction is hard-wired. The illustration below shows a left-top Deskbar in its expanded and contracted states:



The Be Book

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BDeskbar Function Summary

Function Summary

Function	Description
AddItem()	Adds an item to the Deskbar's shelf
CountItems()	Counts items on the Deskbar's shelf
BDeskbar()	Constructor
~Deskbar()	Destructor
Expand()	Expands or contracts the Deskbar
Frame()	Returns the Deskbar's frame in screen coordinates.
GetItemInfo()	Gets information about an item on the Deskbar's shelf
HasItem()	Asks the Deskbar if it has a particular item on its shelf
IsExpanded()	Returns **`true`** if the Deskbar is expanded
Location()	Returns the Deskbar's location
RemoveItem()	Removes an item from the Deskbar's shelf
SetLocation()	Sets the Deskbar's location

Copyright © 2000 Be, Inc. All rights reserved..

System Messages

Declared in: be/app/AppDefs.h
more...

B_MOUSE_WHEEL_CHANGED

Source: The system.
Target: The BWindow of the view the mouse is pointing to, and that of any other view that sets **`B_POINTER_EVENTS`** in its event mask (see **`BView::SetEventMask()`**)
Hook: none

`B_MOUSE_WHEEL_CHANGED` is sent when the mouse wheel is distrubed. The message contains these fields:

Field	Type code	Description
"when"	**`B_INT64_TYPE`**	Event time, in microseconds since 01/01/70
"be:wheel_delta_x"	**`B_FLOAT_TYPE`**	The change in the wheel's horizontal position as a value between -1.0 (left) and 1.0 (right).
"be:wheel_delta_x"	**`B_FLOAT_TYPE`**	The change in the wheel's vertical position as a value

		between -1.0 (down) and 1.0 (up).

B_OBSERVER_NOTICE_CHANGE

Source: `BHandler::SendNotices()`
Target: `Other BHandlers that are watching the notice-sender`
Hook: `none`

B_OBSERVER_NOTICE_CHANGE is sent when the source object (a BHandler) invokes SendNotices(), which broadcasts a notice to all of its "watchers" (other BHandlers). A BHandler can register to be a watcher by calling BHandler::StartWatching().

Field	Type code	Description
B_OBSERVE_WHICH_CHANGE	B_INT32_TYPE	The change that caused the notification to be sent, as taken from an argument to SendNotices().
"be:old_what"	B_INT32_TYPE	The what field from the BMessage that's passed to SendNotices().
Other fields	...	The rest of the fields are copied from the message that's passed to SendNotices().

Channel Control Message

Source: A BChannelControl object
Target: Set through SetTarget() et al.
Hook: none

An *n*-channeled BChannelControl adds the following field to the BControl-defined value-changed message that it sends to its target.

Field	Type code	Description
"be:channel_value" [*n*]	B_INT32_TYPE	The value of each of the object's channels.

This field usurps (but doesn't remove) the "be:value" field that's added by BControl.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Deprecated API

The following API is still supported, but should be replaced as suggested.

Cursor , Constants

Declared in: be/app/AppDefs.h

The **B_HAND_CURSOR** and **B_I_BEAM_CURSOR** constants are deprecated in favor of the new BCursor class.

BDeskbar::AddItem()

Declared in: be/be_apps/Deskbar/Deskbar.h

The BView-accepting version of **AddItem()** has been deprecated:

```
status_t AddItem(BView *view, int32 *id = NULL)  deprecated
```

See AddItem() for more information.

BScreen::ProposeDisplayMode()

Declared in: be/interface/Screen.h

This function is now called **ProposeMode()**.

Media , Kit

media_raw_audio_format is deprecated. Use **media_multi_audio_format** instead.

media_format.u.raw_audio and **media_raw_audio_format::wildcard** have been subsumed by the **media_multi_audio_format** struct, which is upwards compatible.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

OpenGLKit

BGLView

Declared in: be/opengl/GLView
more...

SwapBuffers()

```
void SwapBuffers( bool vSync )
```

In this new version of `SwapBuffers()`, the buffer swap is synchronized with the vertical blanking (if *vSync* is **true**).

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Translation Kit

BTranslationUtils

Declared in: be/translation/TranslationUtils.h
more...

GetBitmap()

```
static BBitmap * GetBitmap(const entry_ref * ref, BTranslatorRoster * use = NULL)
```

New version of `GetBitmap()` takes an `entry_ref` as its initial argument.

Global Functions

Declared in: be/translation/Translator.h

make_nth_translator()

```
BTranslator *make_nth_translator(int32 n, image_id addon, uint32 flags, ...)
```

You implement this function in your translator add-on to create and return a new BTranslator object. A single add-on can create instances from more than one BTranslator subclass. You don't have to publicize how many different translators you can create; instead, you simply create and return an instance of a different subclass for each value of the index *n* (starting with 0) and return **NULL** when you run out of subclasses.

addon is the image that your addon lives in. It's provided as a convenience.

flags is currently unused.

This function replaces the translator add-on API defined in translation/TranslatorAddOn.h.

Translator Add-ons

Declared in: be/translation/TranslatorAddOn.h

All of the functions and symbols in this file have been deprecated. To provide a BTranslator object from an add-on, you implement the make_nth_translator() function, describe immediately above. Translator add-ons themselves still live in the usual places (in the **Translators** subdirectory of `B_*_ADDONS_DIRECTORY`).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslatorRoster

Derived from: (none)
Mix-in classes: BArchivable
Declared in: be/translation/TranslatorRoster.h
Library: libtranslation.so
Summary: *more...*

BTranslatorRoster is the main mechanism through which applications interact with the Translation Kit. An application using the Translation Kit doesn't need to worry about explicitly loading or calling add-ons; BTranslatorRoster transparently handles all the niggling details. The class provides four categories of service: initialization, information, translation, and configuration.

Initialization

You can create an empty BTranslatorRoster (one which has no translators loaded) by instantiating it in the usual fashion:

```
BTranslatorRoster *roster = new BTranslatorRoster;
```

You can then load translators to the newly-created BTranslatorRoster with the `AddTranslators()` method (all paths must be absolute):

```
// load all translators in a given directory
roster->AddTranslators("/boot/home/config/add-ons/viewer/");

// load a specific translator
roster->AddTranslators("/system/add-ons/Translators/photos");
```

More commonly, you want a BTranslatorRoster with the default translators (those in **/boot/home/config/add-ons/Translators**, **/boot/home/config/add-ons/Datatypes**, and **/system/add-ons/Translators**). The static member `Default()` returns just such a beast:

```
BTranslatorRoster *roster = BTranslatorRoster::Default();
```

However, not all BTranslatorRosters are created equal: the object returned by `Default()` is global to the application and controlled by BTranslatorRoster. As a result, you should never `delete` it. Additionally, if you want to load additional translators, you're better off creating a new instance of BTranslatorRoster rather than using the default one:

```
BTranslatorRoster *roster = new BTranslatorRoster;
roster->AddTranslators(NULL); // load default translators
roster->AddTranslators(...); // load additional translators
```

Information

Applications typically ask the following questions of the Translation Kit:

1. Which translators are installed?
2. Which translations can a particular translator carry out?
3. Which translator is best suited for handling my conversion?

An application can determine the installed translators by calling `GetAllTranslators()`. `GetAllTranslators()` returns an array of **`translator_id`** values for the installed translators. A **`translator_id`** is an application-wide value assigned by BTranslatorRoster identifying a specific translator. The following snippet prints out the names of the default translators:

```
BTranslatorRoster *roster = BTranslatorRoster::Default();
int32 num_translators, i;
translator_id *translators;
const char *translator_name, *translator_info;
int32 translator_version;

roster->GetAllTranslators(&translators, &num_translators);
for (i=0;i<num_translators;i++) {
   roster->GetTranslatorInfo(translators[i], &translator_name,
      &translator_info, &translator_version);
   printf("%s: %s (%.2f)n", translator_name, translator_info,
      translator_version/100.);
}

delete [] translators; // clean up our droppings
```

The **`translator_id`** is very valuable; it can be used to query BTranslatorRoster for specific information about a translator's capabilities. This information comes in two nearly identical flavors: `translation_format` and `translator_info`. They are defined in **<be/translation/TranslationDefs.h>**:

```
struct translation_format {
uint32 type;
uint32 group;
float quality;
float capability;
char MIME[251];
char name[251];
}

struct translator_info {
uint32 type;
```

```
translator_id translator;
uint32 group;
float quality;
float capability;
char name[251];
char MIME[251];
}
```

The common fields:

- *group*. Defines the type of media the format represents, i.e. bitmap image, sound, or video. Constants for common media types can be found in **<be/translation/TranslatorFormats.h>**.
- *type*. Type constant defining the specific data format. For example, the type constant for tiff bitmaps differs from the constant for jpeg bitmaps.
- *quality*. Ability of the format to represent data of its media group. This value ranges from a low of 0.0 (utter inability) to a high of 1.0 (encodes all relevant information).
- *capability*. Ability of translator to decode the format. As with the quality, the value ranges from 0.0 (unable to decode) to 1.0 (can decode all variants and extensions).
- *MIME*. MIME type of the format. This is a more reliable indicator of the data format than the *type* field for those formats that have standard MIME names.
- *name*. Human-readable C string describing the format. May include information about the translator as well.

`translator_info` defines an additional field:

- *translator*. **`translator_id`** for the translator associated with the structure.

To find the media formats supported by a particular translator, call `GetInputFormats()` or `GetOutputFormats()` as appropriate. These methods return a list of the supported input or output formats in an array of translation_format.

Many times, however, you don't really care about individual translators; you just want the translator best suited for handling your media stream. In these cases, you can just call `Identify()` on the BPositionIO. BTranslatorRoster will then return, in a `translator_info`, the translator most suited for carrying out the translation. If you are instead interested in finding all the translators capable of handling the data in your stream, use `GetTranslators()` instead.

Note that some translators do not publish their input and output formats. In these cases, `GetInputFormats()` and `GetOutputFormats()` return an empty list of formats. The only way to tell if such a translator supports a particular input or output format is to pass it to `Identify()`.

Translation

Although the function of BTranslatorRoster is to provide translation services, carrying out the translation is simple. All it requires are the input and output **BPositionIO** streams and the type constant for the desired output. In the simplest case, if you know the type constant you want to convert the data into, you can let the BTranslatorRoster decide which translator to use:

```
BPositionIO *in = ..., *out = ...;
BTranslatorRoster *roster = BTranslatorRoster::Default();
uint32 desired_format_constant = ...;

roster->Translate(in, NULL, NULL, out, desired_format_constant);
```

Sometimes, however, you'd like the services of a specific translator. In these cases, you can use the alternate form of **Translate()**:

```
BPositionIO *in = ..., *out = ...;
BTranslatorRoster *roster = BTranslatorRoster::Default();
uint32 desired_format_constant = ...;
translator_id desired_translator_id = ...;

roster->Translate(desired_translator_id, in, NULL, NULL, out,
   desired_format_constant);
```

The Translation Kit won't chain translators for you. If you want to translate from GIF to PNG, you either need a translator that can convert GIF to PNG, or you need to perform two translations: one from GIF to an interchange format, then another from that format to PNG.

Configuration

BTranslatorRoster provides two mechanisms for configuring the behavior of translators: *ioExtension* and **`MakeConfiguationView()`**.

ioExtension is a BMessage that can be passed to most BTranslatorRoster members. It is used by the application to communicate format-specific information to the translator. For example, it could be used to ask a video translator to only translate the first 15 frames of the movie. The *ioExtension* field is also used by the translator to communicate information back to the application. The translator may use it, for example, to tell the application that it is returning a greyscale image.

A translator need not support any particular extension and there is no way for an application to tell if a translator supports any extensions.

A set of standard *ioExtension* messages can be found in **<be/translation/TranslatorFormats.h>** and are explained below:

Name	Type	Direction	Description
B_TRANSLATOR_EXT_HEADER_ONLY	**bool**	app to translator	Only output header if true
B_TRANSLATOR_EXT_DATA_ONLY	**bool**	app to translator	Only output data if true
B_TRANSLATOR_EXT_COMMENT	**string**	n/a	Text comment about data.
B_TRANSLATOR_EXT_TIME	**bigtime_t**	app to translator	If one exists, it specifies a single instant in time. If two exist, they specify a range in time. Time is measured in microseconds.
B_TRANSLATOR_EXT_FRAME	**uint32**	app to translator	Same as kTimeExtension, except the unit of time is frames.
B_TRANSLATOR_EXT_BITMAP_RECT	**BRect**	app to translator	Specifies subsection of a bitmap.
B_TRANSLATOR_EXT_BITMAP_COLOR_SPACE	**uint32**	both	Specifies desired/actual color space of bitmap.
B_TRANSLATOR_EXT_SOUND_CHANNEL	**uint32**	app to translator	Only output specified channel of sound.
B_TRANSLATOR_EXT_SOUND_MONO	**bool**	app to translator	Mix all audio channels to a single mono channel if true.
B_TRANSLATOR_EXT_SOUND_MARKER	**uint32**	both	Specifies markers in sound data. Units are in frames. "1" specifies marker between first sample of last channel and second sample of first channel.
B_TRANSLATOR_EXT_SOUND_LOOP	**uint32**	both	If there is one value, loop from the end of the sample to the loop point. If there are two values, loop from the second value to the first. If there

			are more than two values, then there is a release loop.

BTranslatorRoster contains two methods to facilitate configuration of translators. The first method, **`MakeConfigurationView()`**, instructs the translator to create a BView in which configuration options may be changed. It is the responsibility of the application calling **`MakeConfigurationView()`** to attach the view to a BWindow. The second method, **`GetConfigurationMessage()`**, fills in a BMessage with the current settings for a specified translator. This BMessage can then be passed to **`Translate()`** to request a translation with the settings contained within.

Example

The task of translating data often boils down to finding the right type constant. The following function will print the format type constants associated with a MIME string given a BTranslatorRoster:

```
void find_constant(BTranslatorRoster *roster, const char *mime)
{
   translator_id *translators;
   int32 num_translators;

   roster->GetAllTranslators(&translators, &num_translators);

   for (int32 i=0;i<num_translators;i++) {
      const translation_format *fmts;
      int32 num_fmts;

      roster->GetOutputFormats(translators[i], &fmts, &num_fmts);

      for (int32 j=0;j<num_fmts;j++) {
         if (!strcasecmp(fmts[j].MIME, mime))
            printf("match: %s type %8.8x (%4.4s)n",
               fmts[j].name, fmts[j].type, &fmts[j].type);
      }
   }
}
```

Constructor and Destructor

BTranslatorRoster()

BTranslatorRoster()
BTranslatorRoster(BMessage **model***model***)**

Creates a new container for translators. The no-argument constructor creates an empty container. The BMessage constructor loads all the translators from the directories specified in the **"be:translator_path"** named fields in *model*.

If you just want to load the translators from the standard directories, you can instead use the static member **Default()** described below.

See also: **Default()** static function, **AddTranslators()**

~BTranslatorRoster()

~BTranslatorRoster()

Unloads any translators that were loaded and frees all memory allocated by the BTranslatorRoster.

Static Functions

Default()

```
BTranslatorRoster *Default()
```

This returns a BTranslatorRoster loaded with the default set of translators, loaded from the colon-deliminated list of files and directories found in the environment variable **TRANSLATORS**. If no such variable exists, the translators are loaded from the default locations **/boot/home/config/add-ons/Translators**, **/boot/home/config/add-ons/Datatypes**, and **/system/add-ons/Translators**.

The instance of BTranslatorRoster returned by this function is global to the application and should not be deleted. Although you can add translators to the returned BTranslatorRoster, this is not recommended. It's better to use a new instance of the class.

See also: `AddTranslators()`

Instantiate()

```
BTranslatorRoster *Instantiate(BMessage *from)
```

Returns a new BTranslatorRoster object, allocated by `new` and created with the version of the constructor that takes a BMessage *archive*. However, if the *archive* doesn't contain data for a BTranslatorRoster object, `Instantiate()` returns `NULL`.

See also: `BArchivable::Instantiate()`, `instantiate_object()`, `Archive()`

Version()

```
const char *Version(int32 *outCurVersion,
    int32 *outMinVersion,
    int32 *inAppVersion = B_TRANSLATION_CURRENT_VERSION)
```

Sets *outCurVersion* to the Translation Kit protocol version number and *outMinVersion* to the minimum protocol version number supported. Returns a human-readable string containing version information. Currently, *inAppVersion* must be **B_TRANSLATION_CURRENT_VERSION**.

Member Functions

AddTranslators()

```
virtual status_t AddTranslators(const char *load_path = NULL)
```

Loads all the translators located in the colon-deliminated list of files and directories found in *load_path*. All specified paths must be absolute. If *load_path* is `NULL`, it loads the translators from the locations specified in the **`TRANSLATORS`** environment variable. If the environment variable is not defined, then it loads all the files in the default directories **/boot/home/config/add-ons/Translators**, **/boot/home/config/add-ons/Datatypes**, and **/system/add-ons/Translators**.

RETURN CODES

- `B_OK`. Identification of *inSource* was successful.
- `B_BAD_VALUE`. Error parsing *load_path*.
- Anything else. Error loading add-ons.

See also: `Default()` static function

Archive()

```
virtual status_t Archive(BMessage *into, bool deep = true) const
```

Archives the BTranslatorRoster by recording its loaded add-ons in the BMessage *into*.

See also: `BArchivable::Archive()`, `Instantiate()` static function

GetAllTranslators()

```
virtual status_t GetAllTranslators(translator_id **outList, int32 *outCount)
```

Returns, in *outList*, an array of all the translators loaded by the BTranslationRoster. The number of elements in the array is placed in *outCount*. The application assumes responsibility for deallocating the array.

RETURN CODES

- `B_OK`. Success.
- **`B_NOT_INITIALIZED`**. Internal Translation Kit error.

- **B_BAD_VALUE**. *outList* or *outCount* is **NULL**.

See also: **Identify()**, **GetTranslators()**, **GetAllTranslators()**

GetConfigurationMessage()

virtual status_t **GetConfigurationMessage(translator_id** *forTranslator*, BMessage ***ioExtension*)**

Saves the current configuration information for translator *forTranslator* in *ioExtension*. This information may be flattened, unflattened, and passed to **Translate()** to configure it.

RETURN CODES

- **B_OK**. Success.
- **B_NO_TRANSLATOR**. *forTranslator* not a valid **translator_id**.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.
- **B_BAD_VALUE**. *ioExtension* is **NULL**.
- Anything else. Error passed on from add-on.

See also: **GetConfigurationView()**, **Translate()**

GetInputFormats() , GetOutputFormats()

virtual status_t **GetInputFormats(translator_id** *forTranslator*,
const **translation_format** ***outFormats*,
int32 **outNumFormats*)

virtual status_t **GetOutputFormats(translator_id** *forTranslator*,
const **translation_format** ***outFormats*,
int32 **outNumFormats*)

Returns an array of the published accepted input or output formats for translator *forTranslator*. *outNumFormats* is filled with the number of elements in the array.

RETURN CODES

- **B_OK**. Success.
- **B_NO_TRANSLATOR**. *forTranslator* not a valid **translator_id**.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.
- **B_BAD_VALUE**. *outFormats* or *outNumFormats* is **NULL**.

See also: **GetTranslatorInfo()**

GetTranslatorInfo()

```
virtual status_t GetTranslatorInfo(translator_id forTranslator, const char **outName,
    const char **outInfo, int32 *outVersion)
```

Returns public information about translator *forTranslator*. Sets *outName* with a short description of the translator, *outInfo* with a longer description, and *outVersion* with the translator's version number.

RETURN CODES

- **B_OK**. Success.
- **B_NO_TRANSLATOR**. *forTranslator* not a valid **translator_id**.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.

See also: GetInputFormats(), GetOutputFormats()

GetTranslators()

```
virtual status_t GetTranslators(BPositionIO *inSource,
    BMessage *ioExtension,
    translator_info **outInfo,
    int32 *outNumInfo,
    uint32 inHintType = 0,
    const char *inHintMIME = NULL,
    uint32 inWantType = 0)
```

Identifies the media in *inSource*, returning an array of valid formats and translators equipped to handle them in outInfo. *outNumInfo* holds the number of elements in the array. If *inHintType* or *inHintMIME* is specified, only those translators that can accept data of the specified type are searched. If *inWantType* is specified, only those translators that can output data of that type are searched. *ioExtension* offers an opportunity for the application to specify additional configuration information to the add-ons. The application assumes responsibility for deallocating the array.

RETURN CODES

- **B_OK**. Identification of *inSource* was successful.
- **B_NO_TRANSLATOR**. No suitable translators found.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.
- **B_BAD_VALUE**. *inSource*, *outInfo*, or *outNumInfo* is **NULL**.
- Anything else. Error operating on *inSource*.

See also: `Identify()`, `GetAllTranslators()`

Identify()

```
virtual status_t Identify(BPositionIO *inSource,
    BMessage *ioExtension,
    translator_info *outInfo,
    uint32 inHintType = 0,
    const char *inHintMIME = NULL,
    uint32 inWantType = 0)
```

Identifies the media in *inSource*, returning a best guess of the format and the translator best equipped to handle it in *outInfo*. If *inHintType* or *inHintMIME* is specified, only those translators that can accept data of the specified type are searched. If *inWantType* is specified, only those translators that can output data of that type are searched. *ioExtension* offers an opportunity for the application to specify additional configuration information to the add-ons.

If more than one translator can identify *inSource*, then the one with the highest *quality*capability* is returned.

RETURN CODES

- `B_OK`. Identification of *inSource* was successful.
- **`B_NO_TRANSLATOR`**. No suitable translator found.
- **`B_NOT_INITIALIZED`**. Internal Translation Kit error.
- `B_BAD_VALUE`. *inSource* or *outInfo* is `NULL`.
- Anything else. Error operating on *inSource*.

See also: `GetTranslators()`, `GetAllTranslators()`

MakeConfigurationView()

```
virtual status_t MakeConfigurationView(translator_id forTranslator,
    BMessage *ioExtension,
    BView **outView,
    BRect *outExtent)
```

Returns, in *outView*, a BView containing controls to configure translator *forTranslator*. It is the application's responsibility to attach the BView to a BWindow. The initial size of the BView is given in *outExtent* but may be resized by the application.

RETURN CODES

- `B_OK`. Success.

- **B_NO_TRANSLATOR**. *forTranslator* not a valid **translator_id**.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.
- **B_BAD_VALUE**. *outView* or *outExtent* is NULL.
- Anything else. Error passed on from add-on.

See also: **GetConfigurationMessage()**, **Translate()**

Translate()

```
virtual status_t Translate(BPositionIO *inSource,
    const translator_info *inInfo,
    BMessage *ioExtension,
    BPositionIO *outDestination,
    uint32 inWantOutType,
    uint32 inHintType = 0,
    const char *inHintMIME = NULL)
virtual status_t Translate(translator_id inTranslator,
    BPositionIO *inSource,
    BMessage *ioExtension,
    BPositionIO *outDestination,
    uint32 inWantOutType)
```

These two functions carry out data conversion, converting the data in *inSource* to type *inWantoutType* and placing the resulting output in *outDestination*. *inInfo* should always contain either the output of an **Identify()** call or NULL. The translation uses the translator identified by *inInfo->infoTranslator* or *inTranslator* as appropriate. If *inInfo* is NULL, Translate() will call first **Identify()** to discover the format of the input stream. *ioExtension*, if it is not NULL, provides a communication path between the translator and the application. *inHintType* and *inHintMIME*, if provided, are passed as hints to the translator.

RETURN CODES

- **B_OK**. Success.
- **B_NO_TRANSLATOR**. No suitable translators found.
- **B_NOT_INITIALIZED**. Internal Translation Kit error.
- **B_BAD_VALUE**. *inSource* or *outSource* is NULL.
- Anything else. Error passed on from add-on.

Archived Fields

The `Archive()` function adds the following field to its BMessage argument:

Field	Type code	Meaning
"be:translator_path" (array)	`B_STRING_TYPE`	The location of the Translation Kit add-on (one field per add-on).

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Translation Kit Table of Contents

- The Translation Kit
- BBitmapStream
- Default Media Formats
- BTranslationUtils
- Translator Add-ons
- BTranslator
- BTranslatorRoster

The Be Book **The Translation Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Translation Kit: Master Index

A

Acquire()	BTranslator
AddTranslators()	BTranslatorRoster
Archive()	BTranslatorRoster
Archived Fields	BTranslatorRoster

B

BBitmapStream	BBitmapStream
BBitmapStream()	BBitmapStream
~BBitmapStream()	BBitmapStream

C

Configuration	BTranslatorRoster
Constructor and Destructor	BBitmapStream
Constructor and Destructor	BTranslatorRoster
Constructor and Destructor	BTranslator

D

Default()	BTranslatorRoster
Default Media Formats	Default Media Formats
Default Media Formats	Default Media Formats
DetachBitmap()	BBitmapStream

E

Example	BTranslatorRoster

G

GetAllTranslators()	BTranslatorRoster
GetBitmap()	BTranslationUtils
GetBitmapFile()	BTranslationUtils
GetConfigMessage()	Translator Add-ons
GetConfigurationMessage()	BTranslatorRoster
GetConfigurationMessage()	BTranslator
GetInputFormats()	BTranslatorRoster
GetOutputFormats()	BTranslatorRoster
GetTranslatorInfo()	BTranslatorRoster
GetTranslators()	BTranslatorRoster

I

Identify()	BTranslatorRoster
Identify()	BTranslator
Identify()	Translator Add-ons

Information	BTranslatorRoster
Initialization	BTranslatorRoster
InputFormats()	BTranslator
inputFormats	Translator Add-ons
Instantiate()	BTranslatorRoster

M

MakeConfig()	Translator Add-ons
MakeConfigurationView()	BTranslatorRoster
MakeConfigurationView()	BTranslator
Media Formats	Default Media Formats
Media Formats	The Translation Kit
Member Functions	BBitmapStream
Member Functions	BTranslatorRoster
Member Functions	BTranslator

O

OutputFormats()	BTranslator
outputFormats	Translator Add-ons

P

Position()	BBitmapStream

R

ReadAt()	BBitmapStream
ReferenceCount()	BTranslator
Release()	BTranslator

S

Seek()	BBitmapStream
SetSize()	BBitmapStream
Size()	BBitmapStream
Static Functions	BTranslationUtils
Static Functions	BTranslatorRoster

T

Translate()	BTranslatorRoster
Translate()	BTranslator
Translate()	Translator Add-ons
Translation	BTranslatorRoster
The Translation Kit	The Translation Kit
The Translation Kit	The Translation Kit
BTranslationUtils	BTranslationUtils
BTranslationUtils	BTranslationUtils
translation_format	BTranslatorRoster
Translator Add-on Data	Translator Add-ons
Translator Add-on Functions	Translator Add-ons

Translator Add-ons	Translator Add-ons
Translator Add-ons	Translator Add-ons
BTranslator	BTranslator
BTranslator()	BTranslator
~BTranslator()	BTranslator
TranslatorInfo()	BTranslator
translatorInfo	Translator Add-ons
TranslatorName()	BTranslator
translatorName	Translator Add-ons
BTranslatorRoster	BTranslatorRoster
BTranslatorRoster()	BTranslatorRoster
~BTranslatorRoster()	BTranslatorRoster
TranslatorVersion()	BTranslator
translatorVersion	Translator Add-ons
B_TRANSLATOR_BITMAP	Default Media Formats
B_TRANSLATOR_EX	BTranslatorRoster
translator_info	BTranslatorRoster
B_TRANSLATOR_MEDIA	Default Media Formats
B_TRANSLATOR_MIDI	Default Media Formats
B_TRANSLATOR_NONE	Default Media Formats
B_TRANSLATOR_PICTURE	Default Media Formats
B_TRANSLATOR_SOUND	Default Media Formats
B_TRANSLATOR_TEXT	Default Media Formats

V

Version()	BTranslatorRoster

W

WriteAt()	BBitmapStream

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslator

Derived from: (none)
Mix-in classes: BArchivable
Declared in: be/translation/Translator.h
Library: libtranslation.so
Summary: *more...*

BTranslator is an abstract superclass that you subclass to define your own proprietary translator objects, one translator per BTranslator subclass. You add instances of your BTranslator subclasses to a BTranslatorRoster through **`BTranslatorRoster::AddTranslator()`**. BTranslator objects that your app creates and adds to the Roster are not visible to other applications.

Note that the BTranslator destructor is protected; you never **`delete`** a BTranslator from outside the class. Insead, you **`Release()`** it. See **`Acquire()`** for details.

The primary BTranslator functions are similar to the functions and data that a translator add-on supplies. Most of the BTranslator functions take you to the Translator Add-ons descriptions.

Constructor and Destructor

BTranslator()

```
BTranslator()
```

The constructor must create and return a new instance of your BTranslator subclass. Note that the constructor doesn't `Acquire()` the object it returns.

~BTranslator()

```
protected ~BTranslator()
```

Note that the destructor is protected; you can only delete a BTranslator object from within the implementation of the subclass. From outside the class, you call `Release()`.

Member Functions

Acquire() , Release() , ReferenceCount()

BTranslator ***Acquire**(void)*
BTranslator ***Release**(void)*
int32 **ReferenceCount**(void) *Debugging use only!*

Acquire() and **Release()** increment and decrement the object's reference count. The count starts at 1 (i.e .the object is implicitly acquired when it's created); if the count falls to 0, the object is deleted.

When you add a BTranslator to a BTranslatorRoster, the BTranslator is automatically **Acquire()**'d. When the BTranslatorRoster is deleted, its BTranslators are **Release()**'d. Thus, when you instantiate a BTranslator and add it to a BTranslatorRoster, you and the Roster maintain joint ownership of the object. To give up ownership (such that the BTranslatorRoster will destroy the object when the Roster itself is destroyed), call **Release()** after adding the BTranslator to the Roster.

Acquire() and **Release()** both return a pointer to the BTranslator that was just acquired or released. If **Release()** caused the object to be deleted, it retruns **NULL**.

ReferenceCount(), which returns the current reference count value, is meant for debugging purposes only. **It is not thread-safe!** Don't predicate your code on the value it returns.

ReferenceCount() see Acquire()

GetConfigurationMessage()

virtual status_t **GetConfigurationMessage**(BMessage **ioExtension*)

Optional

Hook function that asks the object to write its current state into the BMessage argument. See GetConfigMessage() [Translator Add-ons] for details.

Identify()

virtual status_t **Identify**(BPositionIO **inSource*, const translation_format **inFormat*, BMessage **ioExtension*,

```
    translator_info *outInfo,
    uint32 outType) = 0
```

Required

Hook function called by the Translator Roster to ask the BTranslator if it knows how to convert *inSource* into the type described by *outType*. See Identify() [Translator Add-ons] for details.

InputFormats() , OutputFormats()

```
virtual const translation_format *InputFormats(int32 *count) const
virtual const translation_format *OutputFormats(int32 *count) const
```

Optional

These functions should be implemented to return arrays of `translation_format` structures that describe the formats that this object supports. If the functions aren't implemented, the object's `Identify()` function will be called each time an application requests a translation. Both functions should set *count* to the number of elements in the format array.

Unlike the analogous translator add-on format arrays, the arrays returned by these functions don't have to be terminated by an empty `translation_format` structure.

MakeConfigurationView()

```
virtual status_t MakeConfigurationView(BMessage *ioExtension,
    BView **outView,
    BRect *outExtent)
```

Optional

Hook function that lets the BTranslator supply a configuration view. See MakeConfig() [Translator Add-ons] for details.

OutputFormats() see InputFormats()

Translate()

```
virtual status_t Translate(BPositionIO *inSource,
    const translator_info *inInfo,
    BMessage *ioExtension,
```

uint32 *outType*,
BPositionIO **outDestination*) = 0

Required

Hook function that asks the BTranslator to translate data from *inSource* to format *outType*, writing the output to *outDestination*. See Translate() [Translator Add-ons] for details.

TranslatorInfo() , TranslatorName() , TranslatorVersion()

virtual const char ***TranslatorInfo**(void) const = 0

virtual const char ***TranslatorName**(void) const = 0

virtual int32 **TranslatorVersion**(void) const = 0

Required

TranslatorInfo() returns a pointer to the translator's long name, e.g. "aiff translator by the Pie Man (pie@the.man)".

TranslatorName() returns a pointer to the translator's short name, e.g. "aiff translator". The short name should be appropriate for display in a menu.

TranslatorVersion() gives an "MM.mm" version number for the translator. For example, a **TranslatorVersion()** of 314 is interpreted as version 3.14.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslator Function Summary

Function Summary

Function	Description
Acquire()	Acquire the BTranslator so it won't be shot in the head behind your back
GetConfigurationMessage()	Hook that fills in a status description
Identify()	Hook that asks the object if it can convert format A to format B
InputFormats()	Hook that asks the object what input formats it knows how to read
MakeConfigurationView()	Hook that asks the object to supply a preferences-style UI
OutputFormats()	Hook that asks the object what output formats it knows how to write
ReferenceCount()	Debugging tool
Release()	Release the BTranslator so it can be deleted
Translate()	Hook that asks the object to translate data
BTranslator()	Constructor
~BTranslator()	Destructor; use Release() instead

TranslatorInfo()	The object's long name
TranslatorName()	The object's short name
TranslatorVersion()	The object's version number

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBitmapStream

Derived from: public BPositionIO
Declared in: be/translation/BitmapStream.h
Library: libtranslation.so
Summary: *more...*

BBitmapStream allows for the easy conversion of a *translator bitmap*, the default Translation Kit bitmap format, to a BBitmap. It's a very limited subclass of BPositionIO capable only of reading and writing translator bitmaps; performing i/o on the object with other types of data can yield strange results. The main attraction of the class is the **DetachBitmap()** method, which returns the contents of the object as a BBitmap.

Most of the time, you won't need to use BBitmapStream directly; BTranslationUtils contains the functionality required to load images from files, resources, or general BPositionIOs into BBitmaps.

The following snippet illustrates typical usage of BBitmapStream *sans* proper error checking:

```
BBitmap *FetchBitmap(char *filename)
{
BFile file(filename, B_READ_ONLY);
BTranslatorRoster *roster = BTranslatorRoster::Default();
BBitmapStream stream;
BBitmap *result = NULL;
if (roster->Translate(&file, NULL, NULL, &stream,
      B_TRANSLATOR_BITMAP) < B_OK)
   return NULL;
stream.DetachBitmap(&result);
return result;
}
```

You can also initialize the class with a BBitmap and use it as input to **BTranslatorRoster::Translate()** to save it in a different format:

```
void StoreTranslatorBitmap(BBitmap *bitmap, char *filename, uint32 type)
{
   BTranslatorRoster *roster = BTranslatorRoster::Default();
   BBitmapStream stream(bitmap); // init with contents of bitmap
   BFile file(filename, B_CREATE_FILE | B_WRITE_ONLY);

   roster->Translate(&stream, NULL, NULL, &file, type);
}
```

Constructor and Destructor

BBitmapStream()

```
BBitmapStream(BBitmap *map = NULL)
```

Creates a new instance of BBitmapStream. If *map* is `NULL`, the stream is initialized to be empty. Otherwise, the BBitmap is converted to a translator bitmap and placed in the stream. The application shares the BBitmap with the BBitmapStream object. It therefore shouldn't `delete` the BBitmap without first calling **`DetachBitmap()`**.

~BBitmapStream()

```
~BBitmapStream()
```

Frees all memory allocated by the BTranslatorRoster.

Member Functions

DetachBitmap()

status_t **DetachBitmap(**BBitmap ***outMap**)

Returns, in *outMap*, a BBitmap representing the image contained in the BBitmapStream. Once `DetachBitmap()` has been called, no further operations should be performed on the BBitmapStream.

RETURN CODES

- **`B_NO_ERROR`**. Success.
- **`B_BAD_VALUE`**. *outMap* is NULL.
- **`B_ERROR`**. There is no BBitmap available.

Position() , ReadAt() , Seek() , SetSize() , WriteAt()

off_t **Position()** const
ssize_t **ReadAt(**off_t *pos*, void **buffer*, size_t **size***)**
off_t **Seek(**off_t *position*, uint32 *whence***)**
status_t **SetSize(**off_t *size***)** const
ssize_t **WriteAt(**off_t *pos*, const void **data*, size_t **size***)**

These methods provide the implementation for the BPositionIO. The class functions identically to BPositionIO with the exception of `ReadAt()` and `WriteAt()`, which read and write only translator bitmaps as described in the class introduction.

See also: **`BPositionIO`**

Size()

off_t **Size()** const

Returns the number of bytes in the translator bitmap in the stream.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBitmapStream Function Summary

Function Summary

Function	Description
BBitmapStream()	Constructor
~BBitmapStream()	Destructor
DetachBitmap()	Detaches BBitmap associated with the object
Position()	Provided for implementation
ReadAt()	Provided for implementation
Seek()	Provided for implementation
SetSize()	Provided for implementation
Size()	Returns the number of bytes in the translator bitmap in the stream
WriteAt()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Default Media Formats

Declared in: be/translation/TranslatorFormats.h
more...

This section describes the default media formats for each of the media groups defined by the Translation Kit.

Media Formats

B_TRANSLATOR_BITMAP

```
struct TranslatorBitmap {
        int32 magic;
        BRect bounds;
        uint32 rowBytes;
        color_space colors;
        uint32 dataSize;
      }
```

`TranslatorBitmap` holds the header information for a translator bitmap. `magic` should always be set to `B_TRANSLATOR_BITMAP`. The uncompressed image data immediately follows the header in the stream. This image data should be precisely `dataSize` bytes long.

All data in the header is stored in big endian format. The data is stored in either big- or little-endian format, depending on the value of the `colors` field.

B_TRANSLATOR_MIDI

This is a standard type 1 MIDI file, as read by and written to the BMidiStore class.

B_TRANSLATOR_MEDIA

As of this writing, a standard for this format has not been finalized, although it will be in a future release.

B_TRANSLATOR_NONE

Obviously, there is no standard format for the no media type.

B_TRANSLATOR_PICTURE

Streams in this format are stored as **`BPicture::Flatten()`**.

B_TRANSLATOR_SOUND

```
struct TranslatorSound {
        int32 magic;
        uint32 channels;
        float sampleFreq;
```

```
    uint32 numFrames;
}
```

TranslatorSound holds the header information for the default sound format. `magic` should always be set to `B_TRANSLATOR_SOUND`. The raw sound data stored in 16 bit linear format, left channel first, immediately follows the header. The data in both the header and the stream are stored in big endian format.

B_TRANSLATOR_TEXT

This is a plain ASCII text file.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiStore

Derived from: public BMidi
Declared in: be/midi/MidiStore.h
Library: libmidi.so
more...

The BMidiStore class defines a MIDI recording and playback mechanism. The MIDI messages that a BMidiStore object receives (at its input) are stored as *events* in an *event list*, allowing a captured performance to be played back later. The object can also read and write—or *import* and *export*—standard MIDI files.

Recording

The ability to record a MIDI performance is vested in BMidiStore's MIDI hook functions (**`NoteOn()`**, **`NoteOff()`**, etc.). When a MIDI hook is invoked, the function fabricates a discrete event based on the data it has received in its arguments, and adds the event to its event list. You don't need to tell a BMidiStore to start recording; it can record from the moment it's constructed.

For example, to record a performance from an external MIDI keyboard, you connect a BMidiStore to a BMidiPort object and then tell the BMidiPort to start:

```
/* Record a keyboard performance. */
BMidiStore MyStore;
BMidiPort MyPort;

MyPort.Open(...);
MyPort.Connect(MyStore);
MyPort.Start();
/* Start playing... */
```

At the end of the performance, you tell the BMidiPort to stop:

```
MyPort.Stop();
```

Timestamps

Events are added to a BMidiStore's event list immediately upon arrival. Each event is given a timestamp as it arrives; the value of the timestamp is the value of the *time* argument that was passed to the MIDI hook function by the "upstream" object's spray function. There's no guarantee that the time arguments of successive MIDI events will be in chronological order. To ensure that the events are properly ordered, you should call **`Sort()`** before you read from the list (note that writing to a MIDI file automatically sorts the list).

BMidiStore's input functions don't call **SnoozeUntil()**: A BMidiStore writes to its event list as soon as it gets a new message, it doesn't wait until the time indicated by the *time* argument.

Erasing and Editing a Recording

You can't. If you make a mistake while you're recording (for example) and want to try again, you can simulate emptying the object by disconnecting the input to the BMidiStore, destroying the object, making a new one, and re-connecting. For example:

Editing the events in the event list is less than impossible (were such a state possible). You can't do it, and you can't simulate it, at least not with the default implementation of BMidiStore. If you want to edit MIDI data, you have to provide your own BMidi-derived class.

Playback

To "play" a BMidiStore's list of events, you call the object's **Start()** function. As described in the BMidi class specification, **Start()** invokes **Run()**. BMidiStore's **Run()** reads events in the order that they appear in the event list, and sprays the appropriate messages to the connected objects. You can interrupt a BMidiStore playback by calling **Stop()**; uninterrupted, the object will stop by itself after it has sprayed the last event in the list.

The events' timestamps are used as the *time* arguments in the spray functions that are called from within **Run()**. But with a twist: The *time* argument that's passed in the first spray call (for a given performance) is always **B_NOW**; subsequent *time* arguments are re-computed to maintain the correct timing in relation to the first event. In other words, when you tell a BMidiStore to start playing, the first event is performed immediately regardless of the actual value of its timestamp.

Setting the Current Event

You can tell the BMidiStore to begin playing from somewhere in the middle of the list by calling **SetCurrentEvent()** before starting the playback. The function takes an index into the list.

If you want to start playing from a particular time offset into the event list, you first have to figure out which event lies at that time. To do this, you ask for the event that occurs at or after the time offset (in milliseconds) through the **EventAtDelta()** function. The value that's returned by this function is suitable as the argument to **SetCurrentEvent()**.

Keep in mind that **EventAtDelta()** returns the index of the first event at *or after* the desired offset. If you need to know the actual offset of the winning event, you can pass its index to **DeltaOfEvent()**:

```
long firstEvent = MyStore->EventAtDelta(3000);
long actualDelta = MyStore->DeltaOfEvent(firstEvent);
```

Reading and Writing MIDI Files

You add events to a BMidiStore's event list by reading, or *importing*, a Standard MIDI File through the **`Import()`** function. You can import any number of files into the same BMidiStore object. After you import a file, the event list is automatically sorted.

One thing you shouldn't do is import a MIDI file into a BMidiStore that contains events that were previously recorded from a BMidiPort (in an attempt to mix the file and the recording). Nor does the reverse work: You can't import a file and *then* record from a BMidiPort. The file's timestamps are incompatible with those that are generated for events that are received from the BMidiPort; the result certainly won't be satisfactory.

To write the event list as a MIDI file, you call BMidiStore's **`Export()`** function:

Constructor and Destructor

BMidiStore() , ~BMidiText()

BMidiStore(void)
virtual ~**BMidiStore()**

The constructor creates a new, empty BMidiStore object. The destructor destroys the object and its storage.

Member Functions

BeginTime()

uint32 **BeginTime**(void) **const**

Returns the time, in ticks, at which the most recent performance started. This function is only valid if the object has actually performed.

CountEvents() , SortEvents() , DeltaOfEvent() , EventAtDelta()

uint32 **CountEvents**(void) **const**
void **SortEvents**(bool *force* = *false*)
uint32 **DeltaOfEvent**(uint32 *index*) **const**
uint32 **EventAtDelta**(uint32 *delta*) **const**

`CountEvents()` returns the number of events in the object's event list.

`SortEvents()` time-sorts the events in the event list. The object maintains a (conservative) notion of whether the events are already sorted; if *force* is **false** and the object doesn't think the operation is necessary, the sorting isn't performed. If force is **true**, the operation is always performed, regardless of its necessity.

`DeltaOfEvent()` returns the "delta time" of the *index*'th event in the object's list of events. An event's delta time is the time span, in milliseconds, between the first event in the event list and itself.

`EventAtDelta()` Returns the index of the event that occurs on or after *delta* milliseconds from the beginning of the event list.

CurrentEvent() see SetCurrentEvent()
DeltaOfEvent() see CountEvents()
ChrisEvert()

Export() , Import()

status_t **Export**(const entry_ref **to_file*, int32 *format*)
status_t **Import**(const entry_ref **from_file*)

Export() ChrisEvert() time-sorts the object's event list, and then writes the list as a standard MIDI file in the designated format; the *format* argument should be either 0 or 1. The file that *to_file* refers to must already exist, and is *not* cleared before it's written.

Import() reads the standard MIDI file from the designated file and adds its events to the object's event list.

RETURN CODES

- B_OK. The file was successfully written or read.
- B_NO_MEMORY (Import() only). Couldn't allocate enough memory to accommodate the file.
- *POSIX errors*. Some other file error prevented the operation.

SetCurrentEvent() , CurrentEvent()

void **SetCurrentEvent(**uint32 *index***)**
uint32 **CurrentEvent(void)**

Sets the object's "current event"—the event that it will perform next—to the event at *index* in the event list.

Returns the index of the event that will be performed next.

SetTempo() , Tempo()

void **SetTempo(**int32 *beatsPerMinute***)**
int32 **Tempo(**void**)** const

Sets and returns the object's tempo—the speed at which it performs events—in beats per minute. The default tempo is 60 bpm.

SortEvents see CountEvents()

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Midi Kit Table of Contents

- The Midi Kit
- BMidi
- BMidiPort
- BMidiStore
- BMidiSynth
- BMidiSynthFile
- BMidiText
- BSamples
- BSynth
- General MIDI Instruments

The Be Book **The Midi Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Midi Kit: Master Index

+

+20062	BMidiSynth

A

AllNotesOff()	BMidiPort
AllNotesOff()	BMidi

B

BeginTime()	BMidiStore

C

Calling MIDI Hooks Directly	BMidi
Channel Pressure	BMidiText
ChannelPressure(), SprayChannelPressure()	BMidi
Close()	BMidiPort
Connect()	BMidi
Connections()	BMidi
Constructor and Destructor	BMidiPort
Constructor and Destructor	BMidiStore

Constructor and Destructor	BMidiSynthFile
Constructor and Destructor	BMidiSynth
Constructor and Destructor	BMidiText
Constructor and Destructor	BMidi
Constructor and Destructor	BSamples
Constructor and Destructor	BSynth
Control Change	BMidiText
ControlChange(), SprayControlChange()	BMidi
CountClients()	BSynth
CountDevices()	BMidiPort
CountEvents()	BMidiStore
Creating a MIDI Filter	BMidi
CurrentEvent()	BMidiStore

D

DeltaOfEvent()	BMidiStore
Disconnect()	BMidi
Duration()	BMidiSynthFile

E

EnableInput()	BMidiSynth
EnableLooping()	BMidiSynthFile
EnableReverb()	BSamples
EnableReverb()	BSynth

Erasing and Editing a Recording	BMidiStore
EventAtDelta()	BMidiStore
Export()	BMidiStore

F

Fade()	BMidiSynthFile
FlushInstrumentCache()	BMidiSynth
Forming Connections	BMidi

G

General MIDI Instruments	General MIDI Instruments
General MIDI Instruments	General MIDI Instruments
Generating MIDI Messages	BMidi
GetAudio()	BSynth
GetDeviceName()	BMidiPort
GetMuteMap()	BMidiSynthFile
GetMuteMap()	BMidiSynth
GetPatches()	BMidiSynthFile
GetSoloMap()	BMidiSynthFile
GetSoloMap()	BMidiSynth

H

Hook Functions	BMidi

I

Import()	BMidiStore
InitCheck()	BMidiPort
Initializing your BMidiSynth	BMidiSynth
Instrument Scope	BMidiSynth
Interpolation()	BSynth
interpolation_mode	BSynth
IsConnected()	BMidi
IsFinished()	BMidiSynthFile
IsInputEnabled()	BMidiSynth
IsLoaded()	BSynth
IsPaused()	BSamples
IsPlaying()	BSamples
IsReverbEnabled()	BSynth
IsRunning()	BMidi

K

KeepRunning()	BMidi
Key Pressure	BMidiText
KeyPressure(), SprayKeyPressure()	BMidi

L

LimiterThreshhold()	BSynth
LoadFile()	BMidiSynthFile

LoadInstrument()	BMidiSynth
LoadSynthData()	BSynth
Loading and Playing	BMidiSynthFile

M

MIDI Hook and Spray Functions	BMidi
MIDI Hook Functions	BMidiPort
MIDI Hook Functions	BMidi
The MIDI Specification	The Midi Kit
MaxSampleVoices()	BSynth
MaxSynthVoices()	BSynth
Member Functions	BMidiPort
Member Functions	BMidiStore
Member Functions	BMidiSynthFile
Member Functions	BMidiSynth
Member Functions	BMidiText
Member Functions	BMidi
Member Functions	BSamples
Member Functions	BSynth
BMidi	BMidi
BMidi()	BMidi
~BMidi()	BMidi
The Midi Kit	The Midi Kit
The Midi Kit	The Midi Kit
BMidiPort	BMidiPort

BMidiPort()	BMidiPort
~BMidiPort()	BMidiPort
BMidiStore	BMidiStore
BMidiStore()	BMidiStore
BMidiSynth	BMidiSynth
BMidiSynth()	BMidiSynth
~BMidiSynth()	BMidiSynth
BMidiSynthFile	BMidiSynthFile
BMidiSynthFile()	BMidiSynthFile
~BMidiSynthFile()	BMidiSynthFile
~BMidiText()	BMidiStore
BMidiText	BMidiText
BMidiText()	BMidiText
~BMidiText()	BMidiText
MuteChannel()	BMidiSynth
MuteTrack()	BMidiSynthFile

N

Note Off	BMidiText
NoteOff(), SprayNoteOff()	BMidi
Note On	BMidiText
NoteOn(), SprayNoteOn()	BMidi

O

Open()	BMidiPort
Output Text	BMidiText

P

Pause()	BMidiSynthFile
Pause()	BSamples
Pause()	BSynth
Percussion Instruments	BMidiSynth
Pitch Bend	BMidiText
PitchBend(), SprayPitchBend()	BMidi
Placement()	BSamples
Playback	BMidiStore
Position()	BMidiSynthFile
Program Change	BMidiText
ProgramChange(), SprayProgramChange()	BMidi

R

Reading and Writing MIDI Files	BMidiStore
Recording a Performance	BSynth
Recording	BMidiStore
RemapInstrument()	BMidiSynth
ResetTimer()	BMidiText
Resume()	BMidiSynthFile
Resume()	BSamples

Resume()	BSynth
Reverb()	BSynth
reverb_mode	BSynth
Run()	BMidi
The Run Thread and Function	BMidiSynthFile
Running Ahead of Time	BMidi
Running and the MIDI Hook Functions	BMidiPort
Running in Real Time	BMidi

S

SampleVolume()	BSynth
sample_exit_hook	BSamples
sample_loop_hook	BSamples
BSamples	BSamples
BSamples()	BSamples
~BSamples()	BSamples
SamplingRate()	BSamples
SamplingRate()	BSynth
ScaleTempoBy()	BMidiSynthFile
Seek()	BMidiSynthFile
SetControllerHook()	BSynth
SetCurrentEvent()	BMidiStore
SetFileHook()	BMidiSynthFile
SetInterpolation()	BSynth
SetPlacement()	BSamples

SetReverb()	BSynth
SetSampleVolume()	BSynth
SetSamplingRate()	BSamples
SetSamplingRate()	BSynth
SetSynthVolume()	BSynth
SetTempo()	BMidiStore
SetTempo()	BMidiSynthFile
SetTransposition()	BMidiSynth
SetVoiceLimits()	BSynth
SetVolume()	BMidiSynth
SetVolume()	BSamples
Setting the Current Event	BMidiStore
SnoozeUntil()	BMidi
SoloChannel()	BMidiSynth
SoloTrack()	BMidiSynthFile
SortEvents()	BMidiStore
Sources and Destinations	BMidi
Spray Functions	BMidi
Spraying Time	BMidi
Start()	BMidiPort
Start()	BMidiSynthFile
Start()	BMidi
Start()	BSamples
Stop()	BMidiPort
Stop()	BMidiSynthFile
Stop()	BMidi

Stop()	BSamples
BSynth and the Audio Server	BSynth
BSynth	BSynth
BSynth()	BSynth
~BSynth()	BSynth
SynthMode()	BSynth
SynthVolume()	BSynth
synth_controller_hook	BSynth
synth_file_hook	BMidiSynthFile
Synthesis Capacity	BSynth
System Common	BMidiText
SystemCommon(), SpraySystemCommon()	BMidi
System Exclusive	BMidiText
SystemExclusive(), SpraySystemExclusive()	BMidi
SystemRealTime()	BMidiText
SystemRealTime(), SpraySystemRealTime()	BMidi

T

Tempo()	BMidiStore
Tempo()	BMidiSynthFile
TempoChange(),SprayTempoChange()	BMidi
Tick()	BMidiSynth
Time	BMidi
Timestamps	BMidiStore
Transposition()	BMidiSynth

U

Unload()	BSynth
UnloadFile()	BMidiSynthFile
UnloadInstrument()	BMidiSynth

V

Volume()	BMidiSynth
Volume()	BSamples

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiSynth

Derived from: BMidi
Declared in: be/midi/MidiSynth.h
Library: libmidi.so
more...

The BMidiSynth class is the MIDI interface to the General MIDI synthesizer. If you want to send MIDI data to the synthesizer, you have to create an instance of BMidiSynth. You can can send MIDI messages to the object directly:

```
/* Create and initialize a BMidiSynth. */
BMidiSynth midiSynth;
midiSynth.EnableInput(true, true);

/* Choose an instrument and play a note. */
midiSynth.ProgramChange(1, B_ACOUSTIC_GRAND);
midiSynth.NoteOn(1, 40, 100, B_NOW);
snooze(1000000);
midiSynth.NoteOff(1, 40, 100, B_NOW);
```

Or you can connect the BMidiSynth to the output of some other BMidi object, such as a BMidiPort:

```
/* Connect the synth to a MIDI port. */
BMidiPort midiPort;
char buf[B_OS_NAME_LENGTH];

/* Initialize the BMidiPort. */
if (midiPort.GetDeviceName(0, buf) == B_OK) {
   midiPort.Open(buf);

   /* midiSynth from above -- already created and initialized. */
   midiPort.Connect(midiSynth);
   midiPort.Start();
   ...
}
```

The one thing you shouldn't do is connect a BMidiFile to a BMidiSynth. If you want to realize the contents of a MIDI file, you should use an instance of BMidiSynthFile instead (BMidiSynthFile is derived from BMidiSynth).

BMidiSynth doesn't spray MIDI messages, so it doesn't do any good to connect other BMidi objects to its output. In other words, don't do this:

```
/* --- DON'T DO THIS --- It's meaningless. */
midiSynth.Connect(someOtherMidiObject);
```

Initializing your BMidiSynth

When you create a BMidiSynth object, it creates an instance of BSynth for you (if the object doesn't already exist). The BSynth object, which its represented globally in your application as **be_synth**, provides control over some of the synthesizer's global parameters, such as volume and reverberation.

Before you send messages to your BMidiSynth, you have to call **EnableInput()**. The function enables the object's input and tells the synthesizer whether it should load the synth file. If you tell **EnableInput()** not to load the file, you'll have to load the instruments that you want yourself. For example, here we load a single instrument, then play a note. We also have to send a **ProgramChange()** message to tell the BMidiSynth object use our loaded instrument on the proper channel (i.e. the channel that we're playing the note on):

```
/* Enable input, but don't load the synth file. */
midiSynth.EnableInput(true, false);

/* Load a couple of instruments. */
midiSynth.LoadInstrument(B_TINKLE_BELL);
midiSynth.LoadInstrument(B_VIOLIN);

/* Associate the instrument with a MIDI channel. */
midiSynth.ProgramChange(1, B_TINKLE_BELL);
midiSynth.ProgramChange(2, B_VIOLIN);

/* Play. */
midiSynth.NoteOn(1, 84, 100);
snooze(10);
midiSynth.NoteOn(2, 60, 100);
snooze(1000000);
midiSynth.NoteOff(1, 84, 100);
snooze(10);
midiSynth.NoteOff(2, 60, 100);
```

The order and number of instruments follow the General MIDI specification, but begin with instrument 0 (some synthesizers and sequencers expect instrument numbers to start at 1). The handy **midi_axe** constants, defined in **be/midi/MidiDefs.h**, provide the descriptive instrument names used here.

Instrument Scope

All of the instrument loading functions (**EnableInput()** and the functions described under **LoadInstrument()**) affect all **be_synth** clients (i.e. all BMidiSynth objects in your application). This can be beneficial: Loading an instrument from one BMidiSynth object means it doesn't have

to be loaded again when, for example, a BMidiSynthFile needs it (the synthesizer is smart about reloading: If an instrument is already loaded, it won't try to load it again). As another example, if you have more than one BMidiSynth object, they can each load all instruments (through **EnableInput()**) without suffering a performance penalty.

However, there's a dark side: If a BMidiSynth unloads an instrument (through **UnloadInstrument()** or **FlushInstrumentCache(false))**, thet instrument disappears for all other BMidiSynth objects as well. You can prevent unwanted instrument unloading by calling **FlushInstrumentCache(true)** (see the function description for more information).

Percussion Instruments

To use the MIDI Channel 10 percussion instruments, you *must* load all instruments:

```
/* I want percussion, therefore... */
midiSynth.EnableInput(true, true);
```

Constructor and Destructor

BMidiSynth()

```
BMidiSynth(void)
```

Creates a new BMidiSynth object. Also constructs a BSynth object and assigns the object to the app-wide **be_synth** variable (if the object doesn't already exist). To turn on your BMidiSynth object, you must call **SetEnabled()** after construction.

~BMidiSynth()

```
virtual ~BMidiSynth()
```

Disconnects the object from the synthesizer, and destroys the object.

Member Functions

EnableInput() , IsInputEnabled()

status_t **EnableInput(**bool *enable*, bool *loadSynthFile***)**
bool **IsInputEnabled(**void**)** const

`EnableInput()` tells the object to connect and disconnect itself to the synthesizer, as *enable* is **true** or **false**. A disabled BMidiSynth doesn't pass on any MIDI messages to the synthesizer.

If *enable* is **true** and *loadSynthFile* is **true**, any instruments in the synth file (as designated by **be_synth**) that haven't already been loaded are immediately loaded. Loading the entire synth file can take a long time (more than a second, less than a minute), and this isn't your last shot at loading instruments, so you may not want to load all at once. If you want to defer loading, you can call **LoadInstrument()** after calling `EnableInput()`.

IsInputEnabled() tells you if the BMidiSynth is currently enabled.

See "Initializing your BMidiSynth" on page30 for more on instrument loading, instrument scope, and the use of Channel 10 percussion.

BMidiSynthFile objects do not have to call `EnableInput()`. Input is automatically enabled when the object loads a file.

RETURN CODES

Currently, **EnableInput()** always return

- B_OK.

FlushInstrumentCache() see LoadInstrument()
GetMuteMap() see MuteChannel()
GetSoloMap() see MuteChannel()
IsInputEnabled() see EnableInput()

LoadInstrument() , RemapInstrument() , UnloadInstrument() , FlushInstrumentCache()

status_t **LoadInstrument(**int16 *instrument***)**
status_t **RemapInstrument(**int16 *from*, int16 *to***)**
status_t **UnloadInstrument(**int16 *instrument***)**
void **FlushInstrumentCache(**bool *flush***)**

These functions affect *all* **be_synth** clients (they should actually be part of the BSynth class).

LoadInstrument() loads the designated instrument from the synth file, if it isn't loaded already. You have to load an instrument before you can use it; if you've already loaded all instruments (through **InputEnabled()**), you won't have to call this function. A loaded instrument is available to all BMidiSynth objects in your application.

RemapInstrument() tells the synthesizer to replace all *from* instrument requests with *to* requests. For example, here we tell the synthesizer to play a gunshot whenever a banjo note is requested:

```
/* Banjo becomes gunshot. A good idea. */
midiSynth.RemapInstrument(B_BANJO, B_GUNSHOT);
```

UnloadInstrument() removes the designated instrument from the synthesizer.

FlushInstrumentCache() has a goofy protocol. If *flush* is **false**, the synthesizer stops playing all current notes (and samples) and unloads all loaded instruments. If *flush* is **true**, the synthesizer's current output is unaffected, but it's prevented from unloading any instruments in the future. In other words, **FlushInstrumentCache(true)** tells the synthesizer to ignore all subsequent **UnloadInstrument()** calls. The inability to unload instruments holds until **FlushInstrumentCache(false)** is called.

See "Initializing your BMidiSynth" on page30 for more on instrument loading, instrument scope, and the use of Channel 10 percussion.

RETURN CODES

Currently, UnloadInstrument() always returns

- B_OK. The other two **status_t** functions return...
- **B_OK**. Instrument successfully loaded or remapped.
- **B_BAD_INSTRUMENT**. Illegal instrument specification.
- B_NO_MEMORY. Not enough memory to load instrument.

MuteChannel() , GetMuteMap() , SoloChannel() , GetSoloMap()

These functions are broken; don't use them.

RemapInstrument() see LoadInstrument()

SetTransposition() , Transposition()

void **SetTransposition(**int16 *transpose***)**
int16 **Transposition(**void**)** const

`SetTransposition()` sets the amount, in halfsteps, by which subsequently generated notes will be transposed (shifted in pitch). Notes that are already sounding are not transposed. The transposition affects all channels of the invoked-upon object.

`Transposition()` returns the current transposition amount (in halfsteps); the default is 0.

SetVolume() , Volume()

void **SetVolume(**double *volume***)**
double **Volume(**void**)** const

`SetVolume()` scales the object's amplitude (on all channels) by *volume*. In addition to affecting subsequently-generated notes, the new volume scale affects notes that are currently being generated by this object.

`Volume()` returns the current amplitude scalar value; the default is 1.0.

SoloChannel() see MuteChannel()

Tick()

uint32 **Tick(**void**)** const

Returns the number of microseconds since the object was created. You should use this measurement for relative values only (for measuring the time between notes, for example). Also, note that the *time* argument to the MIDI hook functions measures time in milliseconds.

Transposition() see SetTransposition()
UnloadInstrument() see LoadInstrument()
Volume() see SetVolume()

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidi

Derived from: *none*
Declared in: be/midi/Midi.h
Library: libmidi.so
more...

BMidi is an abstract class that defines protocols and default implementations for functions that most of the other Midi Kit classes inherit. The functions that BMidi defines fall into four categories:

- *Connection functions*. The connection functions let you connect the output of one BMidi object to the input of another BMidi object.
- *Performance functions*. BMidi objects that generate MIDI data (by reading from a port or file, as examples) implement the **Run()** hook function. **Run()** is the brains of a MIDI performance; **Start()** and **Stop()** control the performance.
- *"Spray" functions*. The "spray" functions send MIDI messages to each of the objects that are connected to the output of the sprayer. There's a spray function for each type of MIDI message; for example, **SprayNoteOn()** corresponds to MIDI's Note On message.
- *"MIDI hook" functions*. The "MIDI hook" functions define the object's response to a particular type of MIDI message. There's a hook function for each MIDI message (**NoteOn()**, **NoteOff()**, etc.). The hook functions are invoked automatically as "upstream" objects call the corresponding spray functions.

Forming Connections

MIDI data streams through your application, passing from one BMidi-derived object to another. Each object does whatever it's designed to do: Sends the data to a MIDI port, writes it to a file, modifies it and passes it on, and so on.

You form a tree of BMidi objects through BMidi's **Connect()** function, which takes a single BMidi argument. By calling **Connect()**, you connect the output of the calling object to the input of the argument object. (i.e. the caller is the source; the argument is the destination.)

Let's say you want to connect a MIDI keyboard to your computer, play it, and have the performance recorded in a file. You connect a BMidiPort object, which reads data from the MIDI port, to a BMidiStore object, which stores the data that's sent to it and can write it to a file:

```
/* Connect the output of a BMidiPort to the input of a BMidiStore. */
BMidiPort m_port;
BMidiStore m_store);
```

```
m_port.Connect(&m_store);
```

You also have to tell the BMidiPort to start listening to the MIDI port, by calling its **Start()** function. This is explained in a later section.

Once you've made the recording, you can play it back by re-connecting the objects in the opposite direction:

```
/* We'll disconnect first, although this isn't strictly necessary. */
m_port.Disconnect(&m_store);
m_store.Connect(&m_port);
```

In this configuration, a **Start()** call to **m_store** would cause its MIDI data to flow into the BMidiPort (and thence to a synthesizer, for example, for realization).

Sources and Destinations

A BMidi object can be the source for any number of other objects:

```
a_midi.Connect(&b_object);
a_midi.Connect(&c_object);
a_midi.Connect(&d_object);
```

A source can get a list of its destinations through the **Connections()** function.

A BMidi object can be the destination for any number of sources:

```
b_midi.Connect(&e_object);
c_midi.Connect(&e_object);
d_midi.Connect(&e_object);
```

However, a destination *can't* get a list of its sources.

Generating MIDI Messages

If your class wants to generate new MIDI data (as opposed to filtering or realizing the data it receives), it must implement the **Run()** function. An implementation of **Run()** should include a `while()` loop that broadcasts one or more MIDI message on each pass (typically it broadcasts only one), by invoking a spray function. To predicate the loop you *must* test the value of the **KeepRunning()** boolean function.

The outline of a **Run()** implementation looks like this:

```
void MyMidi::Run()
{
   while (KeepRunning()) {
      /* Generate a message and spray it. For example... */
```

```
      SprayNoteOn(...);
   }
   /* You MUST exit when KeepRunning() returns false. */
}
```

The **Run()** function *must* exit when **KeepRunning()** returns **false**. If it doesn't, you'll leak threads.

To tell an object to perform its **Run()** function, you call the object's **Start()** function—you never call **Run()** directly. **Start()** causes the object to spawn a thread (its "run" thread) and execute **Run()** within it. When you're tired of the object's performance, you call its **Stop()** function.

Keep in mind that the **Run()** function is only needed in classes that introduce *new* MIDI data into a performance. As examples, BMidiStore's **Run()** sprays messages that correspond to the MIDI data that it stores, and BMidiPort reads data from the MIDI port and sprays messages accordingly.

Another point to keep in mind is that the **Run()** function can run ahead of real time. It doesn't have to generate and spray data precisely at the moment that the data needs to be realized. This is further explained in the section "Time" on page9.

The BMidiSynthFile class differs from the other classes in the way that it implements and uses its **Run()** function; in particular, it doesn't spawn a run thread. See the BMidiSynthFile class for more information.

Spray Functions

The spray functions send data to the BMidi objects that are connected to the running object's output. There's a separate spray function for each of the MIDI message types: **SprayNoteOn()**, **SprayNoteOff()**, **SprayPitchBend()**, and so on. Spray functions are always found within a BMidi's Run() loop, but they can be found in other places as well. For example, if you're creating a MIDI filter, you would use spray functions in the implementations of the object's MIDI hook functions (as explained next).

MIDI Hook Functions

The MIDI hook functions are hooks that are invoked upon an object's connections when the object sprays MIDI data. The functions take the names of the MIDI messages to which they respond: **NoteOn()** responds to a Note On message, **NoteOff()** responds to a Note Off, and so on. For example, this...

```
a_midi.Connect(&b_midi);
a_midi.SprayNoteOn(...);
```

...causes **b_midi**'s **NoteOn()** function to be invoked. The arguments that are passed to **NoteOn()** are taken directly from the **SprayNoteOn()** call.

BMidi doesn't provide default implementations for any of the MIDI hooks; it's up to each BMidi-derived class to decide how to respond to MIDI messages.

Every BMidi object automatically spawns an "inflow" thread when it's constructed. It's in this thread that the spray-invoked MIDI hooks are executed. The inflow thread is always running—the **Start()** and **Stop()** functions don't affect it. As soon as you construct an object, it's ready to receive data.

Calling MIDI Hooks Directly

You can also feed MIDI data to a BMidi object by invoking the MIDI hook functions directly. For example, let's say you just want to play a note on the General MIDI synthesizer. You don't have to create your own BMidi class simply to implement a **Run()** function that sprays the note; instead, all you have to do is this:

```
BMidiSynth midiSynth;

/* Initialize the BMidiSynth as described in that class. */
...
/* Play a note. */
midiSynth.NoteOn(...);
```

Keep in mind that when you invoke a hook function directly, it executes synchronously in the calling thread. Furthermore, the object may also be receiving MIDI messages in its inflow thread. For the Midi Kit-defined classes, this isn't a problem.

Creating a MIDI Filter

Some BMidi objects "act as filters: They receive data, modify it, and then pass it on. To do this, you call the appropriate spray functions from within the implementations of the MIDI hooks. Below is the implementation of the **NoteOn()** function for a proposed class called Transposer. It takes each Note On, transposes it up a half step, and then sprays it:

```
void Transposer::NoteOn(uchar channel, uchar keyNumber,
                  uchar velocity, uint32 time)
{
   uchar new_key = max(keyNumber + 1, 127);
   SprayNoteOn(channel, new_key, velocity, time);
}
```

There's a subtle but important distinction between a filter class and a "performance" class (where the latter is a class that's designed to actually realize the MIDI data it receives). The distinction has to do with time, and is explained in the next section. An implication of the distinction that affects the current discussion is that it may not be a great idea to invest, in a single object, the ability to filter *and* perform MIDI data. Both BMidiStore and BMidiPort are performance classes—objects of these classes realize the data they receive, the former by caching it, the latter by sending it out the MIDI port. In neither of these classes do the MIDI hooks spray data.

Time

Every spray and MIDI hook function takes a final *time* argument. This argument declares when the message that the function represents should be performed. The argument is given in *ticks* (milliseconds). Tick 0 occurs when you boot your computer; the tick counter automatically starts running at that point. To get the current tick measurement, you call the global, Kernel Kit-defined `system_time()` function and divide by 1000 (`system_time()` returns microseconds).

By convention, time arguments are applied at an object's input. In other words, a MIDI hook should look at the time argument, wait until the designated time, and then do whatever it does that it does do. However, this only applies to BMidi-derived classes that are designed to perform MIDI data. Objects that simply filter data *shouldn't* apply the time argument.

To apply the *time* argument, you call the `SnoozeUntil()` function, passing the value of *time*. For example, a "performance" **`NoteOn()`** function would look like this:

```
void MyPerformer::NoteOn(uchar channel, uchar keyNumber,
               uchar velocity, uint32 time)
{
   SnoozeUntil(time);
   /* Perform the data here. */
}
```

If *time* designates a tick that has already tocked, `SnoozeUntil()` returns immediately; otherwise it tells the inflow thread to snooze until the designated tick is at hand.

Spraying Time

If you're implementing the `Run()` function, then you have to generate a time value yourself which you pass as the final argument to each spray function that you call. The value you generate depends on whether your class runs in real time, or ahead of time.

Running in Real Time

If your class conjures MIDI data that needs to be performed immediately, you should use the **`B_NOW`** macro as the value of the *time* arguments that you pass to your spray functions. **`B_NOW`** is simply a cover for (`system_time()`/1000) (converted to **`uint`**). By using **`B_NOW`** as the *time* argument you're declaring that the data should be performed in the same tick in which it was generated. This probably won't happen; by the time the MIDI hooks are called and the data realized, a couple of ticks may have elapsed. In this case, the MIDI hooks' `SnoozeUntil()` calls will see that the time value has passed and will return immediately, allowing the data to be realized as quickly as possible.

Running Ahead of Time

If you're generating data ahead of its performance time, you need to compute the time value so that it pinpoints the correct time in the future. For example, if you want to create a class

that generates a note every 100 milliseconds, you need to do something like this:

```
void MyTicker::Run()
{
   uint32 when = B_NOW;
   uchar key_num;

   while (KeepRunning()) {

      /* Make a new note. */
      SprayNoteOn(1, 60, 64, when);

      /* Turn the note off 99 ticks later. */
      when += 99;
      SprayNoteOff(1, 60, 0, when);

      /* Bump the when variable so the next Note On
      * will be 100 ticks after this one.
      */
      when += 1;
   }
}
```

When a MyTicker object is told to start running, it generates a sequence of Note On/Note Off pairs, and sprays them to its connected objects. Somewhere down the line, a performance object will apply the time value by calling **SnoozeUntil()**.

But what keeps MyTicker from running wild and generating thousands or millions of notes—which aren't scheduled to be played for hours—as fast as possible? It's because the spray functions pass data to the MIDI hooks through Kernel Kit ports that are 1 (one) message deep. So, as long as one of the MIDI hooks calls **SnoozeUntil()**, the spraying object will never be more than one message ahead.

A useful feature of this mechanism is that if you connect a series of BMidi object that *don't* invoke **SnoozeUntil()**, you can process MIDI data faster than real-time. For example, let's say you want to spray data from one BMidiStore object, pass the data through a filter, and then store it in another BMidiStore. The BMidiStore MIDI hooks don't call **SnoozeUntil()**; thus, data will flow out of the first object, through the filter, and into its destination as quickly as possible, allowing you to process hours of real-time data in just a few seconds. Of course, if you add a performance object into this mix (so you can hear the data while it's being processed), the data flow will be tethered, as described above.

Hook Functions

Run()
Contains the loop that generates and broadcasts MIDI messages.

The MIDI hook functions (**NoteOn()**, **NoteOff()**, and so on) are listed in the section "MIDI Hook and Spray Functions" on page15.

Constructor and Destructor

BMidi()

```
BMidi(void)
```

Creates and returns a new BMidi object. The object's inflow thread is spawned and started in this function—in other words, BMidi objects are born with the ability to accept incoming messages. The "run" thread, on the other hand, isn't spawned until **Start()** is called.

~BMidi()

```
virtual ~BMidi()
```

Kills the inflow and run threads after they've gotten to suitable stopping points (as defined below), deletes the list that holds the connections (but doesn't delete the objects contained in the list), then destroys the BMidi object.

The inflow thread is stopped after all currently-waiting MIDI hook messages have been read. No more messages are accepted while the "inflow queue" is being drained. The run thread is allowed to complete its current pass through the run loop and then told to stop (in the manner of the **Stop()** function).

While the destructor severs the connections that this BMidi object has formed, it doesn't sever the connections from other objects to this one. For example, consider the following (improper) sequence of calls:

```
/* DON'T DO THIS... */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
delete b_midi;
```

The `delete` call severs the connection from `b_midi` to `c_midi`, but it doesn't disconnect `a_midi` and `b_midi`. You have to disconnect the object's "back-connections" explicitly:

```
/* ...DO THIS INSTEAD */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
a_midi->Disconnect(b_midi);
delete b_midi;
```

See also: **Stop()**

Member Functions

AllNotesOff()

virtual void **AllNotesOff(**bool *controlMsgOnly*, uint32 *time* = *B_NOW***)**

Sends a B_ALL_NOTES_OFF Control Change message, passing along the timestamp, to all 16 MIDI channels. If *channelMsgOnly* is **false**, the function also sends a Note Off message for all key numbers on all channels.

Connect()

void **Connect(**BMidi **toObject***)**

Connects the BMidi object's output to *toObject*'s input. The BMidi object can connect its output to any number of other objects. Each of these connected objects receives a MIDI hook call as the BMidi sprays messages.

Any object that's been the argument in a **Connect()** call should eventually be disconnected through a call to **Disconnect()**. In particular, care should be taken to disconnect objects when deleting a BMidi object, as described in the destructor.

See also: **~BMidi()**, **Connections()**, **IsConnected()**

Connections()

BList ***Connections(**void**) const**

Returns a BList that contains the objects that are connected to this object's output.

Disconnect()

void **Disconnect(**BMidi **toObject***)**

Severs the BMidi's connection to the argument. The connection must have previously been formed through a call to **Connect()** with a like disposition of receiver and argument.

See also: **Connect()**

IsConnected()

bool **IsConnected(**BMidi **toObject***)** const

Returns **true** if the argument is present in the receiver's list of connected objects.

IsRunning()

```
bool IsRunning(void) const
```

Returns **true** if the object's Run() loop is looping; in other words, if the object has received a Start() function call, but hasn't been told to Stop() (or otherwise hasn't fallen out of the loop).

KeepRunning()

protected:

```
bool KeepRunning(void)
```

Used by the Run() function to predicate its **while()** loop, as explained in the class description. This function should *only* be called from within Run().

Run()

private:

```
virtual void Run(void)
```

A BMidi-derived class places its data-generating machinery in the Run() function, as described in the section "Generating MIDI Messages" on page6. Of particular note: Your implementation of `Run()` must exit when KeepRunning() returns **false**:

```
void MyMidi::Run()
{
   while (KeepRunning()) {
      /* Generate a message and spray it. */
   }
   /* You MUST exit when KeepRunning() returns false. */
}
```

SnoozeUntil()

```
void SnoozeUntil(uint32 tick) const
```

Puts the calling thread to sleep until *tick* milliseconds have elapsed since the computer was booted. This function is meant to be used in the implementation of the MIDI hook functions, as explained in the section "Time" on page9.

Start()

virtual status_t **Start**(void)

Tells the object to spawn its run loop (wherein it executes the **Run()** function) and then immediately returns. You can override this function in a BMidi-derived class to provide your own pre-running initialization. Make sure you call the inherited version of this function within your implementation.

RETURN CODES

- B_OK. The run thread was successfully spawned and resumed.
- *Thread and port error codes*. Something went wrong.

Stop()

virtual void **Stop**(void)

Tells the object to stop generating MIDI data. Calling `Stop()` tells the **KeepRunning()** function to return **false**, thus causing the run loop to terminate. `Stop()` may return before the thread is dead. `This function` doesn't affect the state of in-coming data: The object will still be able to receive MIDI messages through its MIDI hook functions.

You can override this function in a BMidi-derived class to predicate the stop, or to perform post-performance clean-up (as two examples). Make sure, however, that you invoke the inherited version of this function within your implementation.

MIDI Hook and Spray Functions

The protocols for the MIDI hook and spray functions are given below. What a particular function *means* (and the values of its arguments) is defined by the MIDI spec and isn't explained here. See "Spray Functions" and "MIDI Hook Functions" for more information on how these functions are used.

ChannelPressure(), SprayChannelPressure()

```
virtual void  ChannelPressure(uchar channel, uchar pressure, uint32 time = B_NOW)
```

protected:

```
void  SprayChannelPressure(uchar channel, uchar pressure, uint32 time)
```

ControlChange(), SprayControlChange()

```
virtual void  ControlChange(uchar channel, uchar controlNumber,
    uchar controlValue, uint32 time = B_NOW)
```

protected:

```
void  SprayControlChange(uchar channel, uchar controlNumber,
    uchar controlValue, uint32 time)
```

See also: `AllNotesOff()`

KeyPressure(), SprayKeyPressure()

```
virtual void KeyPressure(uchar channel, uchar note,
    uchar pressure, uint32 time = B_NOW)
```

protected:

```
void SprayKeyPressure(uchar channel, uchar note, uchar pressure, uint32 time)
```

NoteOff(), SprayNoteOff()

```
virtual void NoteOff(uchar channel, uchar note,
    uchar velocity, uint32 time = B_NOW)
```

protected:

```
void SprayNoteOff(uchar channel, uchar note, uchar velocity, uint32 time)
```

NoteOn(), SprayNoteOn()

virtual void **NoteOn(**uchar *channel*, uchar *note*, uchar *velocity*,
uint32 *time* = *B_NOW***)**

protected:

void **SprayNoteOn(**uchar *channel*, uchar *note*, uchar *velocity*, uint32 time**)**

PitchBend(), SprayPitchBend()

virtual void **PitchBend(**uchar *channel*, uchar *lsb*, uchar *msb*, uint32 *time* = *B_NOW***)**

protected:

void **SprayPitchBend(**uchar *channel*, uchar *lsb*, uchar *msb*, uint32 *time***)**

ProgramChange(), SprayProgramChange()

virtual void **ProgramChange(**uchar *channel*, uchar *progNum*, uint32 *time* = *B_NOW***)**

protected:

void **SprayProgramChange(**uchar *channel*, uchar *progNum*, uint32 *time***)**

SystemCommon(), SpraySystemCommon()

virtual void **SystemCommon(**uchar *status*, uchar *data1*,
uchar *data2*, uint32 *time* = *B_NOW***)**

protected:

void **SpraySystemCommon(**uchar *status*, uchar *data1*, uchar *data2*, uint32 *time***)**

SystemExclusive(), SpraySystemExclusive()

virtual void **SystemExclusive(**void **data*, size_t *dataLength*,uint32 *time* = *B_NOW***)**

protected:

void **SpraySystemExclusive(**void **data*, size_t *dataLength*, uint32 *time***)**

SystemRealTime(), SpraySystemRealTime()

virtual void **SystemRealTime(**uchar *status*, uint32 *time* = *B_NOW***)**

protected:

void **SpraySystemRealTime(**uchar *status*, uint32 *time***)**

TempoChange(),SprayTempoChange()

void **SprayTempoChange(**int32 *beatsPerMinute*, uint32 *time* = *B_NOW***)**

protected:

void **SprayTempoChange(**int32 *beatsPerMinute*, uint32 *time***)**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidi Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiPort

Derived from: public BMidi
Declared in: be/midi/MidiPort.h
Library: libmidi.so
more...

A BMidiPort object reads and writes MIDI data through a MIDI hardware port. A MIDI hardware port has an input side (MIDI-In) and an output side (MIDI-Out); you can use a single BMidiPort object to communicate with both sides. Also, you can create and use any number of BMidiPort objects in your application—multiple BMidiPort objects can open and use the same hardware port at the same time.

You identify a MIDI port by name, passing it to the BMidiPort constructor or **`Open()`** function. Use the **`GetDeviceName()`** function to retrieve the names of the MIDI ports. The ports are closed through **`Close()`**; they're automatically closed when the BMidiPort object is destroyed.

On a Macintosh, there are two midi ports:

- "midi1" is the modem port.
- "midi2" is the printer port.

On Intel hardware....

A BMidiPort object can only open one port at a time.

Running and the MIDI Hook Functions

- Running a BMidiPort corresponds to MIDI-In: When you tell a BMidiPort to run (through the **`Start()`** function) the object begins reading from MIDI-In and spraying the MIDI messages to its connected objects.
- On the other side, the MIDI hook functions (**`NoteOn()`**, **`NoteOff()`**, etc.) send MIDI messages to MIDI-Out.

It's possible to use the same BMidiPort object to accept data from MIDI-In and broadcast different data to MIDI-Out. You can even connect a BMidiPort object to itself to create a "MIDI through" effect: Anything that shows up at the MIDI-In port will automatically be sent out the MIDI-Out port.

Constructor and Destructor

BMidiPort()

```
BMidiPort(const char *name = NULL)
```

Creates a new BMidiPort object and opens it on the named port. If no name is given, the object remains unopened until `Open()` is called.

~BMidiPort()

```
virtual ~BMidiPort()
```

Closes the connections to the MIDI ports.

Member Functions

AllNotesOff()

bool **AllNotesOff(**bool *controlOnly*, uint32 *time* = *B_NOW***)**

Commands the BMidiPort object to issue an All Notes Off MIDI message to the MIDI-Out port. If *controlOnly* is **true**, only the All Notes Off message is sent. If it's **false**, a Note Off message is also sent for every key number on every channel.

Close()

void **Close(**void**)**

Closes the object's MIDI port. The port should have been previously opened through a call to **Open()**.

CountDevices() , GetDeviceName()

int32 **CountDevices(**void**)**

status_t **GetDeviceName(**int32 *n*, char **name*,
size_t *bufSize* = *B_OS_NAME_LENGTH***)**

These two function work together to let you retrieve the names of all MIDI ports. `CountDevices()` returns the number of MIDI ports that are supported by the machine. `GetDeviceName()` returns, in *name*, the name of the *n*'th MIDI port. *bufSize* is the length of the *name* buffer, in bytes. It needn't be longer than **B_OS_NAME_LENGTH** (defined in OS.h).

RETURN CODES

`GetDeviceName()` returns...

- **B_OK**. Success.
- **B_BAD_VALUE**. *n* is out of range.
- **B_NAME_TOO_LONG**. The device name length is greater than *bufSize*.

InitCheck()

status_t **InitCheck()** const

Returns the status of the previous port-opening call. This function is provided primarily so you can get the status after opening the port through the constructor.

RETURN CODES

- B_OK. The port was successfully opened.
- *POSIX errors*. The open was thwarted.

Open()

```
status_t Open(const char *name)
```

Opens the MIDI port identified by *name*, so the object can read and write MIDI data. Use the **GetDeviceName()** function to get the names of the MIDI ports. The object isn't given exclusive access to the ports that it has opened—other BMidiPort objects, potentially from other applications, can open the same MIDI ports. When you're finished with the ports, you should close them through a (single) call to **Close()**.

The MIDI-Out connection is active from the moment the object is opened: Messages that arrive through the MIDI hook functions are automatically sent to the MIDI-Out port. To begin reading from the MIDI-In port, you have to invoke the object's **Start()** function.

The function returns **B_OK** if the port was successfully opened.

Start() , Stop()

```
virtual status_t Start(void)
virtual void Stop(void)
```

`Start()` tells the object to begin listening to MIDI-In. For each MIDI message that it hears, the object calls the appropriate spray function.

`Stop()` tells the object to stop listening to MIDI-In.

Neither of these functions affects the MIDI-Out side of the port.

MIDI Hook Functions

The BMidiPort class implements the MIDI hook functions to send MIDI data to the MIDI-Out side of the port. For the syntax of the MIDI hook functions, see the BMidi class.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiPort Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiSynthFile

Derived from: public BMidiSynth
Declared in: be/midi/MidiSynthFile.h
Library: libmidi.so
more...

The BMidiSynthFile class reads a standard MIDI file and plays it on the General MIDI synthesizer. Each BMidiSynthFile object can read (and play) only one file at a time. To use a BMidiSynthFile, you create the object, load a MIDI file, and tell the object to **Start()**:

```
/* Create and initialize a BMidiSynthFile. */
BMidiSynthFile midiSynthFile;
entry_ref midiRef;
get_ref_for_path("/boot/optional/midi/QuickBrownFox.mid", &midiRef);
midiSynthFile.LoadFile(&midiRef);

/* Play the file. */
midiSynthFile.Start();
```

You should create a different BMidiSynthFile object for each MIDI file that you want to mix together into a single performance.

In certain circumstances, a single BMidiSynthFile object can load and play more than one MIDI file at the same time, but you shouldn't rely on this feature.

Loading and Playing

When you call **LoadFile()**, the BMidiSynthFile object automatically calls...

```
EnableInput(true, false)
```

It then loads the file's MIDI data into the synthesizer, which loads all the instruments that are needed by the file. If the file uses a lot of different instruments, loading the file can take some time.

When the file is finished playing (either because it's reached the end, or because you called **Stop()**) the instruments are *not* unloaded. This cuts the overhead if you play the file a second time.

The Run Thread and Function

BMidiSynthFile is different from other **Start()**-able BMidi objects in that it doesn't have a run loop. The MIDI data is parsed and realized in the synthesizer's subscriber thread (the thread

that dumps data into the DAC stream). The lack of a run loop shouldn't affect the way you write your code, but you should be aware that the thread isn't there so you won't go looking for it while you're developing your app

Furthermore, BMidiSynthFile doesn't implement the **Run()** function. Starting and stopping the object's performance (activities that are normally handled in the **Run()** function) are handled by the synthesizer in its subscriber thread. If you create a BMidiSynthFile subclass, *don't* try to resurrect the **Run()** function—leave it as a no-op.

As with the BMidiSynth class, the BMidiSynthFile MIDI hook implementations don't call the spray functions. This means that you can't, for example, connect a BMidiSynthFile to a BMidiPort. If you want to play a MIDI file out a MIDI port, use BMidiStore to represent and play the file.

Constructor and Destructor

BMidiSynthFile()

```
BMidiSynthFile(void)
```

Creates a new, empty BMidiSynthFile object. Also constructs a BSynth object and assigns the object to the app-wide **be_synth** variable (if the object doesn't already exist). To load a MIDI file into the BMidiSynthFile object, you must call **LoadFile()** after construction. Unlike plain BMidiSynth instances, however, you *don't* have to call **EnableInput()** (**LoadFile()** calls it for you).

~BMidiSynthFile()

```
virtual ~BMidiSynthFile()
```

Disconnects the object from the synthesizer, unloads the object's file (and all its instruments) and destroys the object.

Member Functions

Duration() , Position() , Seek()

int32 **Duration**(void) const
status_t **Position**(int32 *ticks*)
int32 **Seek**(void)

`Duration()` returns the length of the object's loaded data, measured in 64ths of a MIDI tick.

`Position()` sets the object's current position within the data.

`Seek()` returns the object's current position.

If you want to reposition the "song pointer", you should do it as a percentage of the `Duration()` measurement.

A MIDI tick is defined by the "MIDI Division" field at the bginning of every MIDI file; this value can be specified by the file as either "ppgn" (tempo-dependent) or frames:subframes per second (tempo-independent). The BMidiSynthFile class assumes that the division is tempo-independent.

RETURN CODES

`Currently Position()` always returns

- B_OK.

EnableLooping() see Start()

GetMuteMap() see MuteTrack()

GetPatches()

status_t **GetPatches**(int16 **instruments*, int16 **count*)

Returns an array of the instruments numbers that are needed by the loaded MIDI file. The *instruments* array should be 128 elements long (to be on the safe side), and must be allocated before it's passed in. Upon return, function sets *count* to the number of instrument numbers that it placed in the array. For example:

```
int16 insts[128];
int16 count;
midiSynthFile.GetPatches(insts, &count);
```

```
for (int n = 0; n < count; n++)
   printf("The file uses instrument #%dn", insts[n]);
```

- GetSoloMap() see MuteTrack()
- IsFinished() see Start()

- LoadFile() , UnloadFile()

status_t **LoadFile(**const entry_ref **midiFileRef***)**
void **UnloadFile(**void**)** const

`LoadFile()` tells the object to load, into the synthesizer, the MIDI data from the *midiFileRef*. The synthesizer caches the data for subsequent playback (which you initiate through **`Start()`**). All instruments that are needed to play the file are loaded into the synthesizer, if they're aren't loaded already.

`UnloadFile()` stops playback of the data that was loaded through this object (if it's playing), and flushes the data (removes it from the synthesizer).

In certain circumstances, a single BMidiSynthFile object can load and play more than one MIDI file at the same time, but you shouldn't rely on this feature.

RETURN CODES

- B_OK. The file was found and loaded.
- **`B_BAD_MIDI_DATA`**. *midiFileRef* isn't a MIDI file, or contains corrupt data.
- **`B_NO_MEMORY`**. Not enough memory to load the file.
- *Other POSIX errors*. The file couldn't be opened or read.

- MuteTrack() , GetMuteMap() , SoloTrack() , GetSoloMap()

These functions are broken; don't use them.

- Pause() see Start()
- Position() see Duration()
- Resume() see Start()
- ScaleTempoBy() see Tempo()
- Seek() see Duration()
-

SetFileHook() see Start()

- Set see Tempo()
- SoloTrack() see MuteTrack()

Start() , Stop() , Fade() , Pause() , Resume() , IsFinished() , EnableLooping() , SetFileHook() , synth_file_hook

virtual status_t **Start**(void)
virtual void **Stop**(void)
void **Fade**(void)
void **Pause**(void)
void **Resume**(void)
bool **IsFinished**(void) const
void **EnableLooping**(bool *loop*)
void **SetFileHook**(synth_file_hook *fileDone*, int32 *arg*)
typedef void (***synth_file_hook**)(int32 *arg*)

These functions control the object's performance.

`Start()` tells the synthesizer to start playing the object's loaded MIDI data beginning at the beginning. Note that Start() does *not* halt a performance in progress; in other words, if the object is already playing its data, you'll get a second, performance while the first continues.

`Stop()` immediately halts the currently playing data. **`Fade()`** also stops playback, but is a bit more graceful: It fades out the sound before killing it.

BMidiSynthFile's `Start()` and `Stop()` replace the functions defined by BMidi; in particular, they spawn and control a "run" thread. See "The Run Thread and Function" on page33 for more information.

`Pause()` and `Resume()` do as they say. Note that when you resume playback, "old" notes are not regenerated, but exact timing is respected. For example, lets say you have a MIDI file that contains two notes, one that starts at time 1.0 (seconds) and lasts for 10.0 seconds, and the other starts at time 9.0. You start the file, then **`Pause()`** at time 3.0; as expected, the first note stops. After awhile, you **`Resume()`**; there's a six second silence and then the second note plays.

`IsFinished()` returns **`false`** if the object is currently playing (or paused during play). Note that the function returns **`false`** before you load a file, and returns **`true`** after you've loaded a file but before you begin playing it.

`EnableLooping(`**`true`**`)` tells the object to replay the file when it reaches the end of the data. If the argument is **`false`**, the file isn't replayed. `Stop()` always shuts up playback, even if looping is enabled.

`SetFileHook()` registers a function that's called when the object is finished playing (either because it ran out of data or `Stop()` was called. *arg* is passed to the hook function as its sole argument. Note that the hook function is called when the object is *completely* finished—it isn't called at the end of each pass through the data while looping is enabled.

RETURN CODES

Start() returns...

- B_OK. The data was successfully started.
- **`B_TOO_MANY_SONGS_PLAYING`**. The synthesizer is too busy with other data streams.

Stop() see Start()

synth_file_hook see Start()

Tempo() , SetTempo() , ScaleTempoBy()

int32 **`Tempo(`**void**`)`** const
void **`SetTempo(`**int32 *beatsPerMinute*`)`**
void **`ScaleTempoBy(`**double *scalar*`)`**

`Tempo()` returns the tempo of the data, as read from the MIDI file (or as set by the other functions), in beats-per-minute. The other two functions set the tempo: `SetTempo()` sets it absolutely in beats-per-minute, and `ScaleTempoBy()` scales the current tempo by the argument (*scalar* == 2.0 means the data is played twice as fast, *scalar* == .5 is twice as slow, and so on).

UnloadFile() see LoadFile()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiSynthFile Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSynth

Derived from: (none)
Declared in: be/midi/Synth.h
Library: libmidi.so
more...

The BeOS includes a 16-channel General MIDI software synthsizer designed by HeadSpace Inc. The BSynth class is the interface to the synthesizer itself. Any application that wants to use the synthesizer must include a BSynth object; however, most applications won't need to create the object directly: The BMidiSynth, BMidiSynthFile, and BSamples classes create a BSynth object for you. Furthermore, since BSynth doesn't inherit from BMidi, it doesn't have any API for actually playing MIDI data. To play MIDI data, you need an instance of BMidiSynth or BMidiSynthFile.

An application can have only one BSynth object at a time. The object is represented globally (within your app) as **be_synth**. The classes that create a BSynth for you (BMidiSynth and so on) won't clobber an existing **be_synth**, but the BSynth constructor will.

When it's created, the **be_synth** object tries to find an instrument definition (or "synth") file. This is a file that contains the data (samples and instructions) for creating General MIDI instruments. The BeOS provides two such files (both designed by HeadSpace, and both stored in **B_SYNTH_DIRECTORY**):

- **B_BIG_SYNTH_FILE (/boot/beos/etc/synth/big_synth.sy)** contains 16-bit, 22 kHz data. It takes about 5 Megs of memory when fully loaded.
- **B_LITTLE_SYNTH_FILE (/boot/beos/etc/synth/little_synth.sy)** is 8-bit, 11 kHz data. It's a quarter the size of the big synth, but lacks the big file's fidelity.

The instrument data is read from the file as it's needed. To "pre-load" the entire synth file, use the **BMidiSynth::EnableInput()** function.

BSynth and the Audio Server

The synthesizer produces sound by taking over the Audio Server's DAC stream. It resets the size and number of buffers in the stream, sets the sampling rate, and adds a BSubscriber to the front of the stream. If you want to mix sound files into the MIDI synthesis, you should use the BSamples object rather than add your own DAC stream subscribers. However, if you really want to add your own sample-generating subscribers, *don't* add them to the front of the DAC stream after the **be_synth** subscriber has been added—your subscriber's samples will be clobbered.

The interaction between the synthesizer and the Media Kit will be cleaned up in a subsequent release.

The DAC stream's previous settings are restored when **be_synth** is destroyed.

Synthesis Capacity

The synthesizer can generate up to 32 voices at a time, where a "voice" is either an individual (synthesized) note, or a stream of samples from a BSamples object. By default, it apportions 28 voice "slots" for synthesis and 4 for samples. You can change the settings through the `SetVoiceLimits()` function.

If you ask for more voices than there are voice slots (for example, if you ask for a 29'th note when there are already 28 singing), the synthesizer will try to kill an old note in order to make room for the new note.

There's no guarantee that the synthesizer and DAC stream will have enough time to generate and process everything you ask for, even if you're running below the 32 voice limit. On a lightly loaded, reasonably fast machine, you shouldn't hear any glitches, but a heavy MIDI command stream (for example) could bog it down.

Recording a Performance

There's no API for automatically writing the synthesizer's output to a file. To record a synthesizer performance you have to create your own BSubscriber, add it to the DAC stream (downstream of the synthesizer), and write out the samples that it receives. (See the Media Kit for more information.)

In some cases, the act of recording can be enough of a CPU drag that the synthesizer falls behind realtime (actually, it's the synthesizer's BSubscriber that's getting behind). It may not sound great while you're monitoring the recording, but the data that's written to the file probably won't be affected—the glitches won't be written to the file.

Constructor and Destructor

BSynth()

BSynth(void)
BSynth(synth_mode mode)

Creates and initiailzes a new BSynth object and sets **be_synth** to point to it. The BSynth that **be_synth** currently points to (if any) is deleted. You can only construct one BSynth object per application. Every application that wants to use the synthesizer must have its own BSynth—you can't "share" another application's **be_synth** object. The constructors for the other synthesis classes (BMidiSynth, BMidiSynthFile, and BSamples) create a BSynth for you if one doesn't already exist.

The default constructor sets the following synthesis parameters, shown here with the functions that you can use to reset the values—and that you should refer to for further explanation:

Parameter	Value	Function
Output sampling rate	22 kHz	SetSamplingRate()
Sample interpolation	B_LINEAR_INTERPOLATION	SetInterpolation()
Max synth voices	28	SetVoiceLimits()
Max sample voices	4	SetVoiceLimits()
Limiter threshhold	7	SetVoiceLimits()
Reverb enabled	true	EnableReverb()
Reverb	B_REVERB_BALLROOM	SetReverb()
Synth mode	B_SYNTH_NONE	LoadSynthData()

You must call LoadSynthData() after calling the default constructor to set the synth mode.

The **synth_mode** constructor sets the synthesis parameters (as above) and then sets the synth mode to the argument, one of **B_BIG_SYNTH**, **B_LITTLE_SYNTH**, or **B_SAMPLES_ONLY**. See LoadSynthData() for **synth_mode** definitions.

~BSynth()

virtual ~**BSynth**()

The destructor stops the synthesizer if it's currently playing anything, frees all synthesis-related storage that the BSynth object allocated, and sets **be_synth** to point to **NULL**.

Member Functions

CountClients()

int32 **CountClients**(void) const

Returns the number of synthesis objects (BMidiSynth and BMidiSynthFile) that are actively feeding data to the synthesizer. Note that this count does *not* include BSamples objects.

EnableReverb() , IsReverbEnabled() , SetReverb() , Reverb() , reverb_mode

status_t **EnableReverb**(bool *reverb_enabled*)
bool **IsReverbEnabled**(void) const
void **SetReverb**(reverb_mode *reverb*)
reverb_mode **Reverb**(void) const
typedef enum { **B_REVERB_NONE**, **B_REVERB_CLOSET**, **B_REVERB_GARAGE**, **B_REVERB_BALLROOM**, **B_REVERB_CAVERN**, **B_REVERB_DUNGEON** } **reverb_mode**

`EnableReverb()` turns on and off **be_synth**'s reverberator. `IsReverbEnabled()` returns the current reverberator-enabled state. Reverb is enabled by default.

SetReverb() sets the reverberator's strength. The constants, shown above, are listed in order of increasing "wetness." `Reverb()` returns the current setting. Setting the reverb mode *doesn't* enable the reverberator.

To turn off the reverberator, do this:

```
EnableReverb(false); /* Good *.
```

...rather than:

```
SetReverb(B_REVERB_NONE); /* Bad */
```

RETURN CODES

EnableReverb() returns...

- B_OK. The reverberator was successfully enabled/disabled.
- **B_NO_MEMORY**. Not enough memory to setup the reverberator.

GetAudio()

int32 **GetAudio(**int16 **left*, int16 **right*, int32 *sampleCount*) const

Returns, in *left* and *right*, the last *sampleCount*'th sample frames (split into left and right channels) generated by the synthesizer. Storage for the samples must be allocated by the caller. The function may return fewer samples than requested. The function returns the number of samples that were written into (each of) *left* and *right*.

This function is designed to feed waveform displays (and the like); it isn't intended to be used as a "sound spigot" that you can pipe to a file (for example).

- Interpolation() see SetSamplingRate()
- interpolation_mode see SetSamplingRate()
- IsLoaded() see LoadSynthData()
- IsReverbEnabled() see EnableReverb()
- LimiterThreshhold() see SetVoiceLimits()

- LoadSynthData() , Unload() , SynthMode() , IsLoaded()

status_t **LoadSynthData(**synth_mode *mode***)** status_t **LoadSynthData(**entry_ref **instrument_file***)
void **Unload(**void**)**
synth_mode **SynthMode(**void**)** const
bool **IsLoaded(**void**)** const

LoadSynthData() tells **be_synth** which synth file to use (and unloads the one currently in use, if any). The first version lets you specify the synth file through a synth_mode constant:

- **B_BIG_SYNTH**. Use the big synth file.
- **B_LITTLE_SYNTH**. Use the little synth file.
- **B_SAMPLES_ONLY**. Don't use a synth file, but prepare the object so it can play sampled sounds. You only use this mode if you're only going to use the BSamples object (in other words, if you're not using BMidiSynth or BMidiSynthFile).

If the synthesizer is initialized with a synth file, it will automatically know how to play BSample data.

Currently, **B_SAMPLES_ONLY** doesn't work. You must use one of the other two constants (**B_BIG_SYNTH** or **B_LITTLE_SYNTH**).

The second version lets you set the synth file as an **entry_ref**, thus providing the opportunity to specify a custom synth file. Unfortunately, the synth file format isn't currently public, so you can't create your own synth files (yet).

LoadSynthData() doesn't actually read the instrument definitions from the synth file—in other words, it doesn't really "load" anything. The instruments are loaded as needed during a performance (as specified by a BMidiSynth[File] object). To force instruments to be read, use BMidiSynth's **`EnableInput()`** or **`LoadInstrument()`** function.

`Unload()` stops the synthesizer (if it's currently playing), forgets the instrument file that was used to initialize the synthesizer, and steps out of the audio output mechanism. After you call **`Unload()`**, the **`be_synth`** object is good for nothing until LoadSynthData() is called (whether directly or through a constructor).

`SynthMode()` returns **`be_synth`**'s current synth mode, one of the three modes listed above or **`B_NO_SYNTH`** if the mode hasn't been set.

`IsLoaded()` returns **`true`** if **`be_synth`** has been initialized and is ready to go. Otherwise, it returns **`false`**.

RETURN CODES

`LoadSynthData()` returns...

- **`B_OK`**. **`be_synth`** was successfully initialized.
- **`B_BAD_VALUE`**. Invalid argument.
- **`B_NO_MEMORY`**. Not enough memory to initialize the synthesizer.
- *POSIX errors*. The synth file wasn't found or couldn't be opened.

MaxSampleVoices() see SetVoiceLimits()
MaxSynthVoices() see SetVoiceLimits()

Pause() , Resume()

void **`Pause`**(void)
void **`Resume`**(void)

`Pause()` tells the synthesizer to stop producing sound. It doesn't suspend non-synthesis BMidi objects—in other words, `Pause()` doesn't suspend BMidiPort or BMidiStore objects.

`Resume()` tells the synthesizer to resume producing sound. BMidiSynthFile objects continue reading from where they were paused; BSamples objects start playing from the beginning of their sample data (they *don't* continue from where they were paused).

Reverb() see EnableReverb()
reverb_mode see EnableReverb()
SampleVolume() see SetSynthVolume()

- SamplingRate() see SetSamplingRate()

- SetControllerHook() , synth_controller_hook

void **SetControllerHook(**int16 *controller*, synth_controller_hook *controlHook***)**
typedef void (***synth_controller_hook**)(int16 *channel*, int16 *controller*, int16 *value***)**

Registers a hook function (*controlHook*) that's invoked whenver a MIDI control message is applied to *controller*. The hook function is invoked just after the control message is processed by the synthesizer. The function is passed the channel, controller number, and controller value as taken from the control message.

- SetInterpolation() see SetSamplingRate()
- EnableReverb()
- SetSampleVolume() see SetSynthVolume()

- SetSamplingRate() , SamplingRate() , SetInterpolation() , Interpolation() , interpolation_mode

status_t **SetSamplingRate(**int32 *rate***)**
int32 **SamplingRate(**void**)** const
status_t **SetInterpolation(**interpolation_mode *interp***)**
interpolation_mode **Interpolation(**void**)** const
typedef enum { **B_DROP_SAMPLE**, **B_2_POINT_INTERPOLATION**, **B_LINEAR_INTERPOLATION** } **interpolation_mode**

`SetSamplingRate()` sets the frequency at which **be_synth** produces data, in frames (of audio data) per second. Acceptable rates are 44100, 22050, and 11025; *rate* is rounded to the nearest acceptable value. The default is 22050.

SamplingRate() returns the sampling rate as previously set by `SetSamplingRate()`.

be_synth's sampling rate is independent of the DAC stream's sampling rate. For example, while the default **be_synth** rate is 22050, the default DAC stream rate is 44100. If the two rates don't match, **be_synth**'s BSubscriber object "interpolates" the be_synth data before dumping it into the DAC stream. There are three interpolation schemes, which you set through `SetInterpolation()`:

Constant	Meaning
B_DROP_SAMPLE	Samples are repeated or dropped. It sounds cheap because it *is* cheap.

B_2_POINT_INTERPOLATION	Linear interpolation between adjacent samples. Much better quality, and more expensive, than drop-sample.
B_LINEAR_INTERPOLATION	"Wide" linear interpolation. The best quality, but the most expensive.

`Interpolation()` returns the current interpolation mode setting. The default is `B_LINEAR_INTERPOLATION`.

RETURN CODES

SetSamplingRate() and SetInterpolation() return...

- **B_OK**. The function was successful.
- **B_BAD_VALUE**. Invalid argument.

SetSynthVolume() , SetSampleVolume() , SynthVolume() , SampleVolume()

void **SetSynthVolume(**double *scale***)**
void **SetSampleVolume(**double *scale***)**
double **SynthVolume(**void**)** const
double **SampleVolume(**void**)** const

These functions get and set the master volume scalars for MIDI synthesis and BSamples playback. The scalar is linear: A *scale* of 1.0 (the default) has no affect; a scale of 2.0 multiplies the output by 2.0, and so on. The *scale* value must be at least 0.0 (no gain).

SetVoiceLimits() , MaxSynthVoices() , MaxSampleVoices() , LimiterThreshhold()

status_t **SetVoiceLimits(**int16 *maxSynthVoices*, int16 *maxSampleVoices*, int16 *limiter Threshhold***)**
int16 **MaxSynthVoices(**void**)** const
int16 **MaxSampleVoices(**void**)** const
int16 **LimiterThreshhold(**void**)** const

The synthesizer can generate as many as 32 "voices" simultaneously, where a voice is a MIDI note or a stream of BSamples. The first two arguments tell the synthesizer to set aside some number of voice slots for MIDI synthesis and for samples, respectively; combined, the two arguments mustn't exceed 32. If you ask for too many voices during a performance, the synthesizer will (try to) kill old voices first. By default, the voices are allocated 28 for MIDI synthesis and 4 for samples.

You use the *limiterThreshhold* to estimate the typical voice density (number of simultaneous voices) for a performance. It must be at least 1; the default is 7. The synthesizer uses the value as an amplitude scalar:

- When the voice density during a performance is less than (or equal to) the threshhold (*n*), the dynamic range is "divided" into *n* parts, where each voice gets one part.
- When the voice density (*m*) exceeds the threshhold, the dynamic range is divided into *m* parts.

If you set the value too high (if there are typically fewer simultaneous voices than you estimated) the signal-to-noise ratio will suffer—you'll be dividing the dynamic range into too many (small) parts. If you set it too low and the voice density changes a lot, the balance between voices may become hard to predict and control. A change to the *limiterTreshhold* doesn't affect notes/samples that are currently being produced.

The other three functions return the values that you passed to `SetVoiceLimits()`. Note that these functions don't actually consult the synthsizer—if you pass illegal values to `SetVoiceLimits()`, the querying functions will return those values without complaint.

RETURN CODES

`SetVoiceLimits()` returns...

- `B_OK`. The limits were successfully set.
- `B_BAD_VALUE`. Bad argument value; the previous settings are left unchanged.

- synth_controller_hook see SetControllerHook()
- SynthMode() see LoadSynthData()
- SynthVolume() see SetSynthVolume()
- Unload() see LoadSynthData()

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSynth Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSamples

Derived from: (none)
Declared in: be/midi/Samples.h
Library: libmidi.so
more...

The BSamples class lets you add a stream of audio samples into the General MIDI mix. The main functionality of the class (such as how you define the sound you want to play, whether it should repeat, what speed it should play at, and so on) is embedded in the **`Start()`** function; the class' other functions control and fine-tune the sound as it's playing. To learn how to use a BSamples object, go to the **`Start()`** function.

Theoretically, you can create and play (at the same time) as many BSamples as you want. In practice, you are limited to the sample voice limit set through BSynth's **`SetVoiceLimits()`** function; by default the limit is set to four.

Constructor and Destructor

BSamples()

```
BSamples(void)
```

Creates a new, empty BSamples object. Also constructs a BSynth object and assigns the object to the app-wide **be_synth** variable (if the object doesn't already exist). To initialize and use the BSamples object, invoke its **Start()** function.

Currently, the "samples only" synth mode (B_SAMPLES_ONLY) is broken. Unfortunately, this is the mode used by the BSamples constructor if it has to create a **be_synth** object. The easiest way around this bug is to construct a BMidiSynth or BMidiSynthFile object. If you don't need the object, you can immediately destroy it; the fix is effected by the object's construction.

~BSamples()

```
virtual ~BSamples()
```

Stops the object's playback, calls the "exit hook" function (as set by **Start()**), and destroys the object.

Member Functions

EnableReverb()

void **EnableReverb**(bool *reverb*)

If reverb is true, the object's sound data is sent through the synthesizer's reverberator. To set the reverberation depth (for all reverberated sound), use **BSynth::SetReverb()**.

IsPlaying() see Stop()
Pause() see Stop()
Placement() see SetSamplingRate()
Resume() see Stop()
SamplingRate() see SetSamplingRate()
SetPlacement() see SetSamplingRate()

SetSamplingRate() , SamplingRate() , SetPlacement() , Placement() , SetVolume() , Volume()

void **SetSamplingRate**(double *samplingRate*)
double **SamplingRate**(void) const
void **SetPlacement**(double *stereoPan*)
double **Placement**(void) const
void **SetVolume**(double *volume*)
double **Volume**(void) const

These functions set parameters of the sound data. They can be called while the sound is playing. For details of what these parameters mean, and the values that they accept, see the descriptions of the *samplingRate*, *stereoPan*, and *volume* arguments of the **Start()** function.

SetVolume() see SetSamplingRate()

Start() , sample_loop_hook , sample_exit_hook

void **Start**(void **samples*, int32 *frameCount*, int16 *sampleSize*, int16 *channelCount*, double *sampl* int32 *loopStart*, int32 *loopEnd*, double *volume*, double *stereoPan*, int32 *hookArg*, sample_loop_hook *loopHook*, sample_exit_hook *exitHook*)
typedef bool (***sample_loop_hook**)(int32 *arg*)

typedef void (***sample_exit_hook**)(int32 *arg*)

`Start()` passes in a buffer of audio samples that are immediately played. The playback is performed by the synthesizer in its BSubscriber thread; the `Start()` function itself returns immediately.

The audio data is assumed to be little-endian linear; other parameters of the data (sample size, channel count, etc.) are variable and are declared in the function's arguments. The arguments are:

- *samples* is a pointer to the data itself.
- *frameCount* is the number of frames of audio data in the buffer.
- *sampleSize* is the size of a single sample, in bytes. Currently, the synthesizer only accepts 1- and 2-byte samples.
- *channelCount* is the number of channels of data (1 or 2).
- *samplingRate* is the rate at which you want to the data played back, expressed as frames-per-second; for example, if your sound data was sampled at 44.1kHz and you want to play it back at its "native" pitch, you would set *samplingRate* to 44100.0. The range of valid sampling rate values is [0, ~65 kHz]. The sampling rate is independent of the data's channel count and sample size, and is independent of **be_synth**'s sampling rate. You can change the object's sampling rate on the fly through **SetSamplingRate()**.
- *loopStart* and *loopEnd* specify the first and last frames that are in the "loop section." The loop section can be any valid section of frames within the sound data (i.e. [0, *frameCount* - 1] inclusive). Everything up to the beginning of the loop section is the "attack section"; everything after the loop section is the "release section." When the sound is played, the attack section is heard, then the loop section is repeated until the object is told to **Stop()**, or until the "loop hook" function (defined below) returns **false**, at which point the release section is played. If you don't want the sound to loop, set these *loop...* arguments to 0.

Currently, the release section is automatically faded out over a brief period of time. If your release section is designed to do a slow fade (for example) you probably won't hear it.

- *volume* is an amplitude scalar for this sound. The volume can also be set through **SetVolume()**.
- *stereoPan* locates the sound stereophonically, where -1.0 is hard left, 0.0 is center, and 1.0 is hard right. Notice that if this is a stereo sound, a *stereoPan* value of (say) -1.0 completely attenuates the right channel—it doesn't "move" the right channel into the left channel.
- *hookArg* is an arbitrary value that's passed to the *loopHook* and *exitHook* functions.
- *loopHook* is a hook function that's called each time the loop section is about to repeat. If the function returns **true**, the loop is, indeed, repeated. If it returns **false**, the release

section is played and the sound stops. If you don't supply a *loopHook*, the loop is automatically repeated.

- *exitHook* is called when the sound is all done playing, regardless of how it stopped (whether through `Stop()`, a *hookFunc* return of **`false`**, or because the BSamples object was deleted).

Stop() , Pause() , Resume() , IsPaused() , IsPlaying()

virtual void **`Stop`**`(void)`
void **`Pause`**`(void)`
void **`Resume`**`(void)`
bool **`IsPlaying`**`(void)` const
bool **`IsPaused`**`(void)` const

`Stop()` tells the object to stop playing its sound data, and invokes the exit function, as set through `Start()`.

`Pause()` suspends playback and `Resume()` resumes it from where it was paused.

`Pause()` and `Resume()` are backwards. To pause a sound, call `Resume()`. To resume it, call `Pause()`. Sorry about that.

`IsPlaying()` returns **`true`** if the object is currently playing (paused or not).

`IsPaused()` returns **`true`** if the sound is currently paused.

`IsPaused()` returns `true` if you've called `Pause()`. Thus, since `Pause()` and `Resume()` are backwards, the sense of `IsPaused()` is also backwards.

Volume() see SetSamplingRate()

The Be Book **The Midi Kit** **The Midi Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BSamples Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiSynth Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiText

Derived from: public BMidi
Declared in: be/midi/MidiText.h
Library: libmidi.so
more...

A BMidiText object prints, on standard output, a text description of each MIDI message it receives. You use BMidiText objects to debug and monitor your application; it has no other purpose.

To use a BMidiText object, you construct it and connect it to some other BMidi object as shown below:

```
BMidiText midiText;

otherMidiObj.Connect(midiText);
otherMidiObj.Start();
```

BMidiText's output (the text it displays) is timed: When it receives a MIDI message that's timestamped for the future, the object waits until that time has come before it displays its message. While it's waiting, the object won't process any other in-coming messages. Because of this, you shouldn't connect the same BMidiText object to more than one BMidi object. To monitor two or more MIDI-producing objects, you should connect a separate BMidiText object to each.

The text that's displayed by a BMidiText follows this general format:

timestamp: *MESSAGE TYPE*; *message data*

(Message-specific formats are given in the section "Output Text" on page40.) Of particular note is the *timestamp* field. Its value is the number of milliseconds that have elapsed since the object received its first message. The time value is computed through the use of an internal timer; to reset this timer—a useful thing to do between performances, for example—you call the **ResetTimer()** function.

The BMidiText class doesn't generate or spray MIDI messages, so the performance and connection functions that it inherits from BMidi have no effect.

Constructor and Destructor

BMidiText() , ~BMidiText()

```
BMidiText(void)
virtual ~BMidiText()
```

The constructor creates a new BMidiText object. The object's timer is set to zero and doesn't start ticking until the first message is received. To force the timer to start, call **`ResetTimer(true)`**.

The destructor destroys the object.

Member Functions

ResetTimer()

void **ResetTimer(**bool *start* = false**)**

Sets the object's internal timer to zero. If *start* is **false** the timer doesn't start ticking until the next MIDI message is received. If *start* is **true**, the timer begins immediately.

The timer value is used to compute the timestamp that's displayed at the beginning of each message description.

Output Text

The text strings that a BMidiText object displays are shown below, listed by MIDI event

Channel Pressure

timestamp: CHANNEL PRESSURE; channel = *channel*, pressure = *pressure*

Control Change

timestamp: CONTROL CHANGE; channel = *channel*, control = *ctrl_num*,
value = *ctrl_value*

Key Pressure

timestamp: KEY PRESSURE; channel = *channel*, note = *note*, pressure = *pressure*

Note Off

timestamp: NOTE OFF; channel = *channel*, note = *note*, velocity = *velocity*

Note On

timestamp: NOTE ON; channel = *channel*, note = *note*, velocity = *velocity*

Pitch Bend

timestamp: PITCH BEND; channel = *channel*, lsb = *lsb*, msb = *msb*

Program Change

timestamp: PROGRAM CHANGE; channel = *channel*, program = *program_num*

System Common

timestamp: SYSTEM COMMON; status = *status*, data1 = *data1*, data2= *data2*

System Exclusive

timestamp: SYSTEM EXCLUSIVE;

Followed by the data itself (in hex, byte-by-byte), starting on the next line.

SystemRealTime()

timestamp: SYSTEM REAL TIME; status = *status*

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiText Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

General MIDI Instruments

Declared in: be/midi/MidiDefs.h
more...

The General MIDI Specification declares a set of 128 instruments, numbered 0-127. The Midi Kit provides a set of handy **midi_axe** constants that you can use to identify a MIDI instrument in funtions such as **BMidiSynth::LoadInstrument()** and **BMidi::ProgramChange()**.

#	**Pianos**
0	B_ACOUSTIC_GRAND
1	B_BRIGHT_GRAND
2	B_ELECTRIC_GRAND
3	B_HONKY_TONK
4	B_ELECTRIC_PIANO_1
5	B_ELECTRIC_PIANO_2
6	B_HARPSICHORD
7	B_CLAVICHORD

#	**Tuned Idiophones**
8	B_CELESTA
9	B_GLOCKENSPIEL
10	B_MUSIC_BOX
11	B_VIBRAPHONE

12	B_MARIMBA
13	B_XYLOPHONE
14	B_TUBULAR_BELLS
15	B_DULCIMER

#	**Organs**
16	B_DRAWBAR_ORGAN
17	B_PERCUSSIVE_ORGAN
18	B_ROCK_ORGAN
19	B_CHURCH_ORGAN
20	B_REED_ORGAN
21	B_ACCORDION
22	B_HARMONICA
23	B_TANGO_ACCORDION

#	**Guitars**
24	B_ACOUSTIC_GUITAR_NYLON
25	B_ACOUSTIC_GUITAR_STEEL
26	B_ELECTRIC_GUITAR_JAZZ
27	B_ELECTRIC_GUITAR_CLEAN
28	B_ELECTRIC_GUITAR_MUTED
29	B_OVERDRIVEN_GUITAR
30	B_DISTORTION_GUITAR
31	B_GUITAR_HARMONICS

#	**Basses**
32	B_ACOUSTIC_BASS

33	B_ELECTRIC_BASS_FINGER
34	B_ELECTRIC_BASS_PICK
35	B_FRETLESS_BASS
36	B_SLAP_BASS_1
37	B_SLAP_BASS_2
38	B_SYNTH_BASS_1
39	B_SYNTH_BASS_2

#	**Strings and Timpani**
40	B_VIOLIN
41	B_VIOLA
42	B_CELLO
43	B_CONTRABASS
44	B_TREMOLO_STRINGS
45	B_PIZZICATO_STRINGS
46	B_ORCHESTRAL_STRINGS
47	B_TIMPANI

#	**Ensemble Strings and Voices**
48	B_STRING_ENSEMBLE_1
49	B_STRING_ENSEMBLE_2
50	B_SYNTH_STRINGS_1
51	B_SYNTH_STRINGS_2
52	B_VOICE_AAH
53	B_VOICE_OOH
54	B_SYNTH_VOICE

55	B_ORCHESTRA_HIT

#	**Brass**
56	B_TRUMPET
57	B_TROMBONE
58	B_TUBA
59	B_MUTED_TRUMPET
60	B_FRENCH_HORN
61	B_BRASS_SECTION
62	B_SYNTH_BRASS_1
63	B_SYNTH_BRASS_2

#	**Reeds**
64	B_SOPRANO_SAX
65	B_ALTO_SAX
66	B_TENOR_SAX
67	B_BARITONE_SAX
68	B_OBOE
69	B_ENGLISH_HORN
70	B_BASSOON
71	B_CLARINET

#	**Pipes**
72	B_PICCOLO
73	B_FLUTE
74	B_RECORDER
75	B_PAN_FLUTE

76	B_BLOWN_BOTTLE
77	B_SHAKUHACHI
78	B_WHISTLE
79	B_OCARINA

#	**Synth Leads**	**(Synonyms)**
80	B_LEAD_1	B_SQUARE_WAVE
81	B_LEAD_2	B_SAWTOOTH_WAVE
82	B_LEAD_3	B_CALLIOPE
83	B_LEAD_4	B_CHIFF
84	B_LEAD_5	B_CHARANG
85	B_LEAD_6	B_VOICE
86	B_LEAD_7	B_FIFTHS
87	B_LEAD_8	B_BASS_LEAD

#	**Synth Pads**	**(Synonyms)**
88	B_PAD_1	B_NEW_AGE
89	B_PAD_2	B_WARM
90	B_PAD_3	B_POLYSYNTH
91	B_PAD_4	B_CHOIR
92	B_PAD_5	B_BOWED
93	B_PAD_6	B_METALLIC
94	B_PAD_7	B_HALO
95	B_PAD_8	B_SWEEP

#	**Musical Effects**
96	B_FX_1

97	B_FX_2
98	B_FX_3
99	B_FX_4
100	B_FX_5
101	B_FX_6
102	B_FX_7
103	B_FX_8

#	**Ethnic**
104	B_SITAR
105	B_BANJO
106	B_SHAMISEN
107	B_KOTO
108	B_KALIMBA
109	B_BAGPIPE
110	B_FIDDLE
111	B_SHANAI

#	**Percussion**
112	B_TINKLE_BELL
113	B_AGOGO
114	B_STEEL_DRUMS
115	B_WOODBLOCK
116	B_TAIKO_DRUMS
117	B_MELODIC_TOM
118	B_SYNTH_DRUM

119	B_REVERSE_CYMBAL

#	Sound Effects
120	B_FRET_NOISE
121	B_BREATH_NOISE
122	B_SEASHORE
123	B_BIRD_TWEET
124	B_TELEPHONE
125	B_HELICOPTER
126	B_APPLAUSE
127	B_GUNSHOT

The Be Book The Midi Kit The Midi Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Midi Kit

The Musical Instrument Digital Interface (MIDI) is a standard for representing and communicating musical data. Its fundamental notion is that instantaneous musical events generated by a digital musical device can be encapsulated as "messages" of a known length and format. These messages can then be transmitted to other computer devices where they're acted on in some manner.

The Midi Kit understands the MIDI software format (including Standard MIDI Files). With the Kit, you can create a network of objects that generate and broadcast MIDI messages. Applications built with the Midi Kit can read MIDI data that's brought into the computer through a MIDI port, process the data, write it to a file, and send it back out through the same port. The Kit also contains a General MIDI synthesizer that you can use to realize your MIDI scores. This is a software synthesizer that includes reverberation—you don't any outboard equipment to use it.

The documentation of the Midi Kit is divided into two parts: The first part describes the basic MIDI classes; the second part describes the classes that give you access to the Headspace synthesizer. There are four basic MIDI classes:

- The BMidi class is the centerpiece of the Kit. It defines the tenets to which all MIDI-processing objects adhere, and provides much of the machinery that realizes these ideas. BMidi is abstract—you never create direct instances of the class. Instead, you construct and

connect instances of the other Kit classes, all of which derive from BMidi. You can also create your own classes that derive from BMidi.

- BMidiPort knows how to read MIDI data from and write it to a MIDI hardware port.
- BMidiStore provides a means for storing MIDI data, and for reading, writing, and performing Standard MIDI Files.
- BMidiText is a debugging aid that translates MIDI messages into text and prints them to standard output. You should only need this class while you're designing and fine-tuning your application.

The synthesizer classes are:

- BSynth represents the General MIDI synthesizer and controls some of its global settings (reverb, volume, and so on). Every app that wants to use the synthesizer must have a single BSynth object (represented by the global **be_synth** variable). The other synthesis classes (below) create this object for you.
- BMidiSynth is a BMidi-derived class that connects the basic MIDI world (as represented by the BMidi class) to the synthesizer.
- BMidiSynthFile is a subclass of BMidiSynth that lets you play a MIDI file on the synthesizer.
- BSamples is an interface to the synthesizer's sound data stream. It lets you mix normal audio data in with the

synthesized sound.

The MIDI Specification

This documentation doesn't attempt to teach you anything about the MIDI or General MIDI specifications. In many cases, you don't need to know anything about the specs (or at least not much). For copies of the MIDI specs, search the Web for "midi specification" (there are dozens of copies out there); or go to the official source at http://www.midi.org.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMidiStore Summary

The Midi Kit Table of Contents The Midi Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslationUtils

Derived from: none
Declared in: be/translation/TranslationUtils.h
Library: libtranslation.so
Summary: *more...*

BTranslationUtils contains a set of static functions which load bitmap images. The functions use the Translation Kit to convert the data from its native format to a **BBitmap** via a **BBitmapStream**, the upshot of which is that if you can convert the data to a translator bitmap, you can successfully use these functions. The image may come from a file, resource, or **BPositionIO**. The application is always responsible for deallocating the returned BBitmap.

Static Functions

GetBitmap()

```
static BBitmap *GetBitmap(const char *name, BTranslatorRoster *use = NULL)
static BBitmap *GetBitmap(entry_ref *ref, BTranslatorRoster *use = NULL)
static BBitmap *GetBitmap(uint32 type, int32 id, BTranslatorRoster *use = NULL)
static BBitmap *GetBitmap(uint32 type, const char * name, BTranslatorRoster *use = NULL)
static BBitmap *GetBitmap(BPositionIO *stream, BTranslatorRoster *use = NULL)
```

The first version of the function returns the image held in the file *name* if it exists. Otherwise, it returns the application resource of type **B_TRANSLATOR_BITMAP** named *name*. The second version does the same thing, but uses an entry_ref, *ref*, to locate the file.

The third and fourth versions of the function search for the specified resource and return the bitmap contained therein.

The final version returns the bitmap found in *stream*. This form of the function is particularly useful when used with **BMemoryIO**.

In all cases, *use* specifies the BTranslatorRoster to use in translating the image. If *use* is NULL, the default translator (as returned by **BTranslatorRoster::Default()**) is used.

If the file or stream doesn't contain a suitable bitmap, the functions return NULL.

See also: **BTranslatorRoster::Default()**

GetBitmapFile()

```
static BBitmap *GetBitmapFile(const char *name, BTranslatorRoster *use = NULL)
```

Returns the image held in the file *name*. If the file doesn't contain a suitable image, the function returns NULL. *use* specifies the BTranslatorRoster used to translate the image. If *use* is NULL, the default translator (as returned by **BTranslatorRoster::Default()**) is used.

See also: **BTranslatorRoster::Default()**

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslationUtils Function Summary

Function Summary

Function	Description
GetBitmap()	Loads a bitmap from a resource or BPositionIO
GetBitmapFile()	Returns a bitmap contained in a file

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMemoryIO

Derived from: public BPositionIO
Declared in: be/support/DataIO.h
Library: libbe.so
Summary: *more...*

BMemoryIO represents a buffer of memory. You assign the buffer to a BMemoryIO object upon construction.

Constructor and Destructor

BMemoryIO()

```
BMemoryIO(void *buffer size_t numBytes)
BMemoryIO(const void *buffer size_t numBytes)
```

The BMemoryIO constructor assigns the object a *buffer* with at least *numBytes* of available memory. If the buffer is declared `const`, the object is read-only; calls to **`Write()`** and **`WriteAt()`** will fail. Otherwise, the buffer can be both read and written. In either case, the caller retains responsibility for the buffer; the BMemoryIO object won't free it.

~BMemoryIO()

```
virtual ~BMemoryIO()
```

The BMemoryIO destructor does nothing.

Member Functions

Position() see Seek()

ReadAt()

```
virtual ssize_t ReadAt(off_t position, void *buffer, size_t numBytes)
```

Copies up to *numBytes* bytes of data from the object into the *buffer* and returns the actual number of bytes placed in the buffer. The data is read beginning at the *position* offset.

This function doesn't read beyond the end of the data. If there are fewer than *numBytes* of data available at the *position* offset, it reads only through the last data byte and returns a smaller number than *numBytes*. If *position* is out of range, it returns 0.

Seek() , Position()

```
virtual off_t Seek(off_t position, int32 mode)
virtual off_t Position(void) const
```

`Seek()` sets the position in the data buffer where the `Read()` and `Write()` functions (inherited from BPositionIO) begin reading and writing. How the *position* argument is understood depends on the *mode* flag. There are three modes:

- **SEEK_SET**. The *position* passed is an offset from the beginning of allocated memory; in other words, the current position is set to *position*. For this mode, *position* should be a positive value.
- **SEEK_CUR**. The *position* argument is an offset from the current position; the value of the argument is added to the current position.
- SEEK_END. The *position* argument is an offset from the end of the object's data buffer. In this mode the *position* argument must be negative.

`Write()` fails if the current position is beyond the end of assigned memory.

Both `Seek()` and `Position()` return the current position as an offset in bytes from the beginning of allocated memory.

WriteAt()

```
virtual ssize_t WriteAt(off_t position, const void *buffer, size_t numBytes)
```

Copies *numBytes* bytes of data from *buffer* into allocated memory beginning at *position*, and returns the number of bytes written.

`WriteAt()` won't write outside the memory buffer. If *position* is beyond the end of the buffer, it returns 0. If the object is read-only, it returns `B_NOT_ALLOWED`.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMemoryIO Function Summary

Function Summary

Function	Description
BMemoryIO()	Constructor
~BMemoryIO()	Destructor
Position()	Provided for implementation
ReadAt()	Provided for implementation
Seek()	Provided for implementation
WriteAt()	Provided for implementation

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Translator Add-ons

Declared in: be/translation/TranslatorAddOn.h
Library: libtranslation.so
more...

This section describes the functions and data that a translator add-on must (or can) supply. The translator add-ons that you create and install are loaded into a BTranslatorRoster object through its `Default()` or `AddTranslators()` function.

You compile your add-on as a shared library, link against **libtranslation.so**, and install the add-on object file in the **Translators** subdirectory of `B_USER_ADDONS_DIRECTORY`.

The table below briefly lists the translator add-on symbols:

Symbol	Required?	Description
`translatorInfo`	yes	long description of translator
`translatorName`	yes	short name of translator
`translatorVersion`	yes	translator version number
`Identify()`	yes	identify data in a BPositionIO
`Translate()`	yes	translate data from one format to another
`inputFormats`	no	list of supported input formats
`outputFormats`	no	list of supported output formats
`GetConfigMessage()`	no	save current configuration in a BMessage
`MakeConfig()`	no	create a view for user-configuration of translator

Translator Add-on Functions

GetConfigMessage()

```
status_t GetConfigMessage(BMessage *ioExtension)
```

Optional

This function stores the current configuration in the *ioExtension* in such a manner that it may be flattened, unflattened, and then passed to **Translate()** as an *ioExtension*.

Identify()

```
status_t Identify(BPositionIO *inSource,
    const translation_format *inFormat,
    BMessage *ioExtension,
    translator_info *outInfo,
    uint32 outType)
```

Required

If the translator understands how to convert the data contained in *inSource* to media type *outType*, it fills *outInfo* with details about the input format and return **B_OK**. If it doesn't know how to translate the data, it returns **B_NO_TRANSLATOR**.

The *quality* and *capability* fields in *outInfo* are used by the Translation Kit in selecting the best suited translator for a given translation, so it is best to be conservative in choosing these values.

If the media format doesn't have a type code in **<be/support/TypeConstants.h>** (as will most likely be the case), choose a reasonable value. For example, the Targa handler included with the BeOS distribution uses the type code 'TGA '.

The translator need not fill in the *translator* field in *outInfo*; BTranslatorRoster fills in this value for you.

Remember that *inSource* may be arriving from any source, including a live network feed. It's therefore best to steer clear of calls such as **BPositionIO::Size()** which force all the data out of the stream. Similarly, it is good practice to read only as much of the stream as you need.

inFormat, if it is not `NULL`, is provided as a hint to the format of the data in *inSource*. Since it is only a hint, the data may very well be in some other format.

ioExtension, if it is not `NULL`, contains additional information for the add-on. It is described at length in the section in BTranslatorRoster titled "Configuration".

outType may be zero, in which case any output format is acceptable.

MakeConfig()

```
status_t MakeConfig(BMessage *ioExtension,
    BView **outView,
    BRect *outExtent)
```

Optional

This function creates a new BView through which the user configures the add-on. For example, it could be used to control the degree of image compression used or the video frame rate. The bounds of the view are returned in *outExtent*, although it can be resized at will by an external source. Changes to the configuration take effect immediately, although translations should be carried out with the same parameters throughout.

Translate()

```
status_t Translate(BPositionIO *inSource,
    const translator_info *inInfo,
    BMessage *ioExtension,
    uint32 outType,
    BPositionIO *outDestination)
```

Required

The translator translates data from *inSource* to format *outType*, writing the output to *outDestination*. *outType* may be zero, in which case it is assumed to be the default format type for the media group. As in **`Identify()`**, *inInfo* serves as a hint to the format of the data in *inSource*. *ioExtension* fills its usual role as a container of configuration information. The function returns **`B_OK`** if it's able to convert the data successfully. If it's unable to do so, it returns either **`B_NO_TRANSLATOR`** or an error value as appropriate.

Translator Add-on Data

inputFormats , outputFormats

translation_format **inputFormats**[];
translation_format **outputFormats**[];

Optional

These arrays tell BTranslatorRoster which formats the add-on supports. If they are not exported by the translator, the add-on's **Identify()** will be called each time an application requests a translation. Each array ends in an empty **translation_format** structure, so a typical *inputFormats* would look like:

```
translation_format inputFormats[] = {
{ 'TGA ', B_TRANSLATOR_BITMAP, 0.6, 0.5, "image/targa",
"Targa bitmap format" },
{ B_TRANSLATOR_BITMAP, B_TRANSLATOR_BITMAP, 0.4, 0.6, "image/x-be-bitmap",
"Be Bitmap format" },
{ 0, 0, 0, 0, 0, 0 }
};
```

and similarly for *outputFormats*.

translatorInfo , translatorName , translatorVersion

char **translatorInfo**[];
char **translatorName**[];
int32 **translatorVersion**;

Required

translatorInfo returns a pointer to the translator's long name, e.g. "aiff translator by the Pie Man (pie@the.man)".

translatorName returns a pointer to the translator's short name, e.g. "aiff translator". The short name should be appropriate for display in a menu.

translatorVersion gives an "MM.mm" version number for the translator. For example, a **TranslatorVersion()** of 314 is interpreted as version 3.14.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Translation Kit

The Translation Kit provides a framework for converting data streams between different media formats. A word processor, for example, could use the Translation Kit to import and export documents in a variety of formats, including HTML, PostScript, and plain ASCII, while working in its own native format.

The translation itself is performed by translator add-ons (or, less globally, BTranslator objects). You can create your own translator add-ons and BTranslator subclasses, or you can use the translators that are supplied by Be and third parties.

The Translation Kit operates on **`BPositionIO`** objects and their descendents. Since **`BFile`** derives from BPositionIO, the Translation Kit is naturally suited for importing and exporting files. However, it's far more general; the Translation Kit could be used, for example, to translate input directly to or from a network connection, provided a suitable BPositionIO subclass was available. More importantly, **`BMemoryIO`** and **`BMallocIO`** can be used to load data directly into or out of memory.

Applications primarily interact with the Translation Kit through the BTranslatorRoster class. This class encapsulates the functionality required to load and unload translators, discover their capabilities, configure the translators, and execute the translations.

Applications may also use the BTranslationUtils class, a collection of static utility functions designed to simplify access to the Translation Kit. Currently the class only defines members for loading bitmap images from files, resources, and general BPositionIO data streams.

Media Formats

The Translation Kit identifies a particular media format by a *media group* and *format type*. Media groups identify the class of media represented by a format. Definitions for the common media groups can be found in **<be/translation/TranslatorFormats.h>**:

Media Group	Constant
Bitmap	`B_TRANSLATOR_BITMAP`
BPicture data	`B_TRANSLATOR_PICTURE`
Text	`B_TRANSLATOR_TEXT`
Sound	`B_TRANSLATOR_SOUND`
Standard MIDI	`B_TRANSLATOR_MIDI`
Streaming media	`B_TRANSLATOR_MEDIA`
nothing	`B_TRANSLATOR_NONE`

A format type is a type constant identifying the specific data format, i.e. tiff, aiff, mpeg. The standard type constants can be found in **<be/support/TypeConstants.h>**.

It is often convenient to define a baseline format to which all translators of a given media group adhere. An application that understands the baseline format could then use all the translators for that group. This also simplifies the job of the translator, which minimally needs only to translate between the baseline and the new formats. Well-behaved translators will always be able to translate to and from the default media format (if one exists for its media group).

The Translation Kit defines a default media format for each existing media group. They are explained at length in the section "Default Media Formats". The format type constant for each default format is the same as the media group constant; for example, the format type for the default bitmap format is **B_TRANSLATOR_BITMAP**.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BTranslatorRoster Function Summary

Function Summary

Function	Description
AddTranslators()	Adds translators to the object
Archive()	Archives a BTranslatorRoster into a BMessage
Default()	Returns a BTranslatorRoster loaded with the default set of translators
GetAllTranslators()	Returns the translators loaded by the object
GetConfigurationMessage()	Returns the current configuration information for a given translator
GetInputFormats()	Returns the output formats published by a given translator
GetOutputFormats()	Returns the input formats published by a given translator
GetTranslatorInfo()	Returns public information about a given translator
GetTranslators()	Returns translators equipped to translate a given BPositionIO
Identify()	Identifies the format of a data stream and the translator best equipped to handle it
Instantiate()	Creates a BTranslatorRoster from a BMessage

MakeConfigurationView()	Returns a BView for configuring a given translator
Translate()	Translation work-horse
BTranslatorRoster()	Constructor
~BTranslatorRoster()	Destructor
Version()	Returns version information about the installed Translation Kit

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Support Kit

System Beep

Declared in: be/support/Beep.h
more...

Two new system beep functions:

- `add_system_beep_event()` lets you register a "system beep event." A system beep event is registered and identified by name.
- `system_beep()` plays the soundfile that's associated with the named system beep event.

The user is responsible for associating a system beep event with a soundfile; this is done through the **Sounds** preference. You can associate an event with a soundfile programmatically by using a BMediaFiles object; note that the API defined in **Beep.h** doesn't help in this regard.

Currently, the BeOS defines three system beep events: "Startup", "Beep", and "New E-mail". If you pass `NULL` to `system_beep()`, the "Beep" sound is played.

The old `beep()` function is supported, but deprecated. `beep()` is now a cover for

```
system_beep(NULL)
```


The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFiles

Derived from: none
Declared in: be/media/MediaFiles.h
Library: libmedia.so
Allocation: Constructor only
Summary: *more...*

One feature provided by the Media Kit is the ability to assign media files (sounds and graphics files, for example) to named elements, which can then be used to locate user-configured sound and graphics options.

That's the technical way of saying that the Media Kit provides the ability to assign sounds and bitmap graphics to events and system attributes, so you can configure the appearance and behavior of your BeOS computer's user interface.

Identifying an Entry

Each entry in the media files registry consists of three elements: a **type**, an **item name**, and an **entry_ref**. The type is the type of media data the entry represents. For example, this could be "sound" or "bitmap." The item name is the actual name of the entry in the registry, such as "Startup" or "desktop image." The entry_ref identifies the file that's been assigned to that particular entry.

An application can instantiate a BMediaFiles object and then use the **GetRefFor()** function to find out what file is assigned to a particular registry entry. For instance, if your application needs to access the desktop image file, you can get this information as follows:

```
entry_ref ref;

if (GetRefFor("bitmap", "desktop image", &ref) == B_OK) {
   /* have your way with the desktop image file */
}
```

The user uses the Sounds preference application to assign sound files to events, such as the system beep and the startup sound. These are named "Beep" and "Startup" respectively. The **beep()** function will always play whatever sound is assigned to the Beep event.

Hook Functions

- GetNextRef()
- GetNextType()
- GetRefFor()
- RemoveRefFor()
- RewindRefs()
- RewindTypes()
- SetRefFor()

Constructor and Destructor

BMediaFiles()

```
BMediaFiles(void)
```

Constructor.

~BMediaFiles()

```
virtual ~BMediaFiles(void)
```

Destructor.

Member Functions

GetNextRef()

```
virtual status_t GetNextRef(BString *outType,
    entry_ref *outRef = NULL)
```

Returns the next media type in *outType*. These types include things like "sound" and "bitmap."

RETURN CODES

- B_OK. No error.
- **B_BAD_INDEX**. No more entry_refs to get.
- **B_BAD_VALUE**. *outType* is **NULL**.

GetNextType()

```
virtual status_t GetNextType(BString *outType)
```

Returns the next media type in *outType*. These types include things like "sound" and "bitmap."

RETURN CODES

- B_OK. No error.
- **B_BAD_INDEX**. No more types to get.

GetRefFor() , SetRefFor() , RemoveRefFor()

```
virtual status_t GetRefFor(const char *type,
    const char *item,
    entry_ref *outRef )
```

```
virtual status_t SetRefFor(const char *type,
    const char *item,
    entry_ref &outRef )
```

```
virtual status_t RemoveRefFor(const char *type,
    const char *item,
    entry_ref &outRef )
```

`GetRefFor()` returns the entry_ref for the media file that's been assigned to the specified type and item name pair. For example, if you wanted to retrieve the sound file that's been assigned as the system startup sound, you might specify *type* as "sound" and *item* as "startup sound."

`SetRefFor()` sets, by entry_ref, the file to be used for the specified *type*/*item* pairing.

`RemoveRefFor()` removes the specified entry_ref for the given *type*/*item* pairing.

RETURN CODES

- B_OK. No error.
- **`B_ENTRY_NOT_FOUND`**. (`GetRefFor()` only) the specified type/item pair isn't assigned.

RemoveRefFor() see GetRefFor()

RewindRefs()

virtual status_t **`RewindRefs(`**const char **type**`)`

Rewinds the entry_ref list for the specified media type, so you can start scanning from the beginning of the list.

RETURN CODES

- B_OK. No error.
- **`B_BAD_VALUE`**. *type* is **`NULL`**.
- Other errors. Unable to rewind; the Media Server may not be responding.

RemoveItem()

virtual status_t **`RemoveItem(`**const char **type**`)`

Removes an entrire entry from the media files database. Unlike **`RemoveRefFor()`**, which removes an entry_ref from an entry, this function actually deletes the entire entry.

RETURN CODES

- B_OK. No error.
- Other errors. Unable to remove the item; the Media Server may not be responding.

RewindTypes()

virtual status_t **`RewindTypes(`**void`)`

Rewinds to the beginning of the media type list. If you've been using a BMediaFiles object and want to begin scanning again from the beginning of the media type list, you should call this function.

RETURN CODES

- B_OK. No error.
- Other errors. Unable to rewind; the Media Server may not be responding.

SetRefFor() see GetRefFor()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaFiles Function Summary

Function Summary

Function	Description
GetNextRef()	Returns the next entry_ref for the specified media type
GetNextType()	Returns the next media type in the registry
GetRefFor()	Returns the entry_ref for the specified type and item name
BMediaFiles()	Constructor
~BMediaFiles()	Destructor
RemoveItem()	Removes an entire entry from the media files database
RemoveRefFor()	Removes the entry_ref for the specified type and item name
RewindRefs()	Rewinds to the beginning of the entry_ref list for the specified media type
RewindTypes()	Rewinds to the beginning of the media type list
SetRefFor()	Sets the entry_ref for the specified type and item name

---

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BParameterWeb Function Summary

Function Summary

Function	Description
CountGroups()	Returns the number of groups attached to the BParameterWeb
CountParameters()	Returns the number of BParameters attached to the BParameterWeb
GroupAt()	Returns the group at the specified offset into the list of groups in the BParameterWeb
MakeGroup()	Creates a new BParameterGroup and attaches it to the BParameterWeb
Node()	Returns the media_node for the BControllable that owns this BParameterWeb
ParameterAt()	Returns the BParameter at the specified index
BParameterWeb()	Constructor
~BParameterWeb()	Destructor

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Thread and Team Concepts

A thread is a synchronous process that executes a series of program instructions. A team is a group of threads that make up a single program or application.

Every application has at least one thread: When you launch an application, an initial thread—the *main thread*—is automatically created (or *spawned*) and told to run. The main thread executes the ubiquitous **`main()`** function, winds through the functions that are called from **`main()`**, and is automatically deleted (or *killed*) when **`main()`** exits.

The Be operating system is *multithreaded*: from the main thread you can spawn and run additional threads; from each of these threads you can spawn and run more threads, and so on. All the threads in all applications run concurrently and asynchronously with each other.

Threads are independent of each other. Most notably, a given thread doesn't own the other threads it has spawned. For example, if thread A spawns thread B, and thread A dies (for whatever reason), thread B will continue to run. (But before you get carried away with the idea of leap-frogging threads, you should take note of the caveat in "Death and the Main Thread".)

Threads and the POSIX fork() function are not compatible. You can't mix calls to **`spawn_thread()`** (the function that creates a new thread) and **`fork()`** in the same application: If you call **`spawn_thread()`** and then try to call **`fork()`**, the **`fork()`** call will fail. And vice versa.

Although threads are independent, they do fall into groups called *teams*. A team consists of a main thread and all other threads that "descend" from it (that are spawned by the main thread directly, or by any thread that was spawned by the main thread, and so on). Viewed from a higher level, a team is the group of threads that are created by a single application. You can't "transfer" threads from one team to another. The team is set when the thread is spawned; it remains the same throughout the thread's life.

All the threads in a particular team share the same address space: Global variables that are declared by one thread will be visible to all other threads in that team.

Spawning a Thread

You spawn a thread by calling the **`spawn_thread()`** function. The function assigns and returns a system-wide **`thread_id`** number that you use to identify the new thread in subsequent function calls. Valid **`thread_id`** numbers are positive integers; you can check the success of a spawn thus:

```
thread_id my_thread = spawn_thread(...);

if ((my_thread) < B_OK)
   /* failure */
else
   /* success */
```

The arguments to **spawn_thread()**, which are examined throughout this description, supply information such as what the thread is supposed to do, the urgency of its operation, and so on.

Threads and App Images

A conceptual neighbor of spawning a thread is the act of loading an executable (or loading an *app image*). This is performed by calling the **load_image()** function. Loading an image causes a separate program, identified as a file, to be launched by the system. For more information on the **load_image()** function, see Images.

Telling a Thread to Run

Spawning a thread isn't enough to make it run. To tell a thread to start running, you must pass its **thread_id** number to either the **resume_thread()** or **wait_for_thread()** function:

- **resume_thread()** starts the new thread running and immediately returns. The new thread runs concurrently and asynchronously with the thread in which **resume_thread()** was called.
- **wait_for_thread()** starts the thread running but doesn't return until the thread has finished. (You can also call **wait_for_thread()** on a thread that's already running.)

Of these two functions, **resume_thread()** is the more common means for starting a thread that was created through **spawn_thread()**. **wait_for_thread()** is typically used to start the thread that was created through **load_image()**.

The Thread Function

When you call **spawn_thread()**, you must identify the new thread's *thread function*. This is a global C function (or a static C++ member function) that the new thread will execute when it's told to run. The thread function, defined as **thread_func**, takes a single (**void ***) argument and returns an **int32** error code. When the thread function exits, the thread is automatically killed.

You pass a thread function as the first argument to **spawn_thread()**. For example, here we spawn a thread that uses a function called **lister()** as its thread function. The last argument to spawn_thread() is forwarded to the thread function:

```
int32 lister(void *data)
{
   /* Cast the argument. */
   BList *listObj = (BList *)data;
   ...
}

int32 main()
{
   BList *listObj = new BList();
```

```
    thread_id my_thread;

    my_thread = spawn_thread(lister, ..., (void *)listObj);
    resume_thread(my_thread);
    ...
}
```

See Passing Data to a Thread for other methods of passing data to a thread.

Thread Names

A thread can be given a name which you assign through the second argument to **spawn_thread()**. The name can be 32 characters long (as represented by the **B_OS_NAME_LENGTH** constant) and needn't be unique—more than one thread can have the same name.

You can look for a thread based on its name by passing the name to the **find_thread()** function; the function returns the **thread_id** of the so-named thread. If two or more threads bear the same name, the **find_thread()** function returns the first of these threads that it finds.

You can retrieve the **thread_id** of the calling thread by passing **NULL** to **find_thread()**:

```
thread_id this_thread = find_thread(NULL);
```

To retrieve a thread's name, you must look in the thread's **thread_info** structure. This structure is described in the **get_thread_info()** function description.

Dissatisfied with a thread's name? Use the **rename_thread()** function to change it. Fool your friends.

Thread Priorities

In a multi-threaded environment, the CPUs must divide their attention between the candidate threads, executing a few instructions from this thread, then a few from that thread, and so on. But the division of attention isn't always equal: You can assign a higher or lower *priority* to a thread and so declare it to be more or less important than other threads.

You assign a thread's priority (an integer) as the third argument to **spawn_thread()**. There are two categories of priorities: "time-sharing" and "real-time."

- **Time-sharing (*values from 1 to 99*)**. A time-sharing thread is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.
- **Real-time (*100 and greater*)**. A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highest priority is executed first. The thread is allowed to run without being preempted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

The Kernel Kit defines seven priority constants (see Thread Priority Values for the list). Although you can use other, "in-between" value as the priority argument to `spawn_thread()`, it's suggested that you stick with these.

Furthermore, you can call the `suggest_thread_priority()` function to let the Kernel Kit determine a good priority for your thread. This function takes information about the thread's scheduling and CPU needs, and returns a reasonable priority value to use when spawning the thread.

Synchronizing Threads

There are times when you may want a particular thread to pause at a designated point until some other (known) thread finishes some task. Here are three ways to effect this sort of synchronization:

- The most general means for synchronizing threads is to use a semaphore. The semaphore mechanism is described in great detail in Semaphores.
- Synchronization is sometimes a side-effect of sending data between threads. This is explained in "Passing Data to a Thread", and in Ports.
- Finally, you can tell a thread to wait for some other thread to die by calling `wait_for_thread()`, as described earlier.

Controlling a Thread

There are four ways to control a thread while it's running:

- You can put the calling thread to sleep for some number of microseconds through the `snooze()` and `snooze_until()` functions.
- You can suspend the execution of any thread through the `suspend_thread()` function. The thread remains suspended until you "unsuspend" it through a call to `resume_thread()` or `wait_for_thread()`.
- You can send a POSIX "signal" to a thread through the **`send_signal()`** function. The **`SIGCONT`** signal tries to unblock a blocked or sleeping thread without killing it; all other signals kill the thread. To override this behavior, you can install your own signal handlers.
- You can kill the calling thread through `exit_thread()`, or kill some other thread through `kill_thread()`. Feeling itchy? Try killing an entire team of threads: The **`kill_team()`** function is more than a system call. It's therapy.

Death and the Main Thread

As mentioned earlier, the control that's imposed upon a particular thread isn't visited upon the "children" that have been spawned from that thread. However, the death of an application's main thread can affect the other threads:

When a main thread dies, the game is pretty much over. The main thread takes the team's

heap, its statically allocated objects, and other team-wide resources—such as access to standard IO—with it. This may seriously cripple any threads that linger beyond the death of the main thread.

It's possible to create an application in which the main thread sets up one or more other threads, gets them running, and then dies. But such applications should be rare. In general, you should try to keep your main thread around until all other threads in the team are dead.

Passing Data to a Thread

Every thread has a *message cache*. You can write to a thread's message cache through the `send_data()` function. The thread can pick up your message (a combination of an integer and a buffer) through `receive_data()`. The cache is only one message deep; if there's a message already in the cache, send_data will block. Conversely, if there's no message in the cache, receive_data() will block.

You can also pass data to thread through a port. Arbitrarily deep, ports are more flexible than the message cache. See Ports for details.

The Be Book The Kernel Kit The Kernel Kit Index

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Kernel Kit

The Kernel Kit is a collection of C functions that let you define and control the contexts in which your application operates. There are five main topics in the Kit:

- "Threads and Teams". A thread is a synchronous computer process. By creating multiple threads, you can make your application perform different tasks at (virtually) the same time. A team is the collection of threads that your application creates.

- "Ports". A port can be thought of as a mailbox for threads: A thread can write a message to a port, and some other thread (or, less usefully, the same thread) can then retrieve the message.

- "Semaphores". A semaphore is a system-wide counting variable that can be used as a lock that protects a piece of code. Before a thread is allowed to execute the code, it must acquire the semaphore that guards it. Semaphores can also be used to synchronize the execution of two or more threads.

- "Areas". The area functions let you allocate large chunks of virtual memory. The two primary features of areas are: They can be locked into the CPU's on-chip memory, and the data they hold can be shared between applications.

- "Images". An image is compiled code that can be dynamically linked into a running application. By loading and unloading images you can make run-time decisions about the resources that your application has access to. Images are of particular interest to driver designers.

The rest of this chapter describes these topics in detail. The final three sections...

- "System Information",
- "Time Information",
- and "Miscellaneous Functions and Constants"

...fill in the gaps.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Semaphore Concepts

A semaphore acts as a key that a thread must acquire in order to continue execution. Any thread that can identify a particular semaphore can attempt to acquire it by passing its `sem_id` identifier—a system-wide number that's assigned when the semaphore is created—to the `acquire_sem()` function. The function blocks until the semaphore is actually acquired.

An alternate function, `acquire_sem_etc()` lets you specify the amount of time you're willing to wait for the semaphore to be acquired, and let you acquire the semaphore more than once in a single go. Unless otherwise noted, characteristics ascribed to `acquire_sem()` apply to `acquire_sem_etc()` as well.)

When a thread acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other threads. The semaphore remains unavailable until it's passed in a call to the `release_sem()` function.

The code that a semaphore "protects" lies between the calls to `acquire_sem()` and `release_sem()`. The disposition of these functions in your code usually follows this pattern:

```
if (acquire_sem(my_semaphore) == B_NO_ERROR) {
   /* Protected code goes here. */
   release_sem(my_semaphore);
}
```

Keep in mind that...

- The calls to the acquire and release functions needn't be locally balanced (although this is by far the most common use). A semaphore can be acquired within one function and released in another. Acquisition and release of the same semaphore can even be performed by two different threads.
- Checking the value returned by `acquire_sem()` is *extremely* important. If an acquire-blocked thread is unblocked by a signal (a return o f `B_INTERRUPTED`), the thread shouldn't procede to the critical section.

The Thread Queue

Every semaphore has its own *thread queue*: This is a list that identifies the threads that are waiting to acquire the semaphore. A thread that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's thread queue where it sits blocked in the `acquire_sem()` call. Each call to `release_sem()` umblocks the thread at the head of that semaphore's queue, thus allowing the thread to return from its call to `acquire_sem()`.

Semaphores don't discriminate between acquisitive threads—they don't prioritize or otherwise reorder the threads in their queues—the oldest waiting thread is always the next to acquire the semaphore.

The Thread Count

To assess availability, a semaphore looks at its *thread count*. This is a counting variable that's initialized when the semaphore is created. Ostensibly, a thread count's initial value (which is passed as the first argument to `create_sem()`) is the number of threads that can acquire the semaphore at a time. (As we'll see later, this isn't the entire story, but it's good enough for now.) For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1—in other words, only one thread can acquire the semaphore at a time.

An initial thread count of 1 is by far the most common use; a thread count of 0 is also useful. Other counts are much less common.

Calls to `acquire_sem()` and `release_sem()` alter the semaphore's thread count: `acquire_sem()` decrements the count, and `release_sem()` increments it. When you call `acquire_sem()`, the function looks at the thread count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function returns immediately.
- If the count is zero or less, the semaphore is unavailable, and the thread is placed in the semaphore's thread queue.

The initial thread count isn't an inviolable limit on the number of threads that can acquire a given semaphore—it's simply the initial value for the sempahore's thread count variable. For example, if you create a semaphore with an initial thread count of 1 and then immediately call `release_sem()` five times, the semaphore's thread count will increase to 6. Furthermore, although you can't initialize the thread count to less-than-zero, an initial value of zero itself is common—it's an integral part of using semaphores to impose an execution order (as demonstrated later).

Summarizing the description above, there are three significant thread count value ranges:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* `acquire_sem()` calls will return without blocking.
- If the count is 0, there are no queued threads, but the next `acquire_sem()` call will block.
- A negative count (*-n*) means there are *n* threads in the semaphore's thread queue, and the next call to `acquire_sem()` will block.

Although it's possible to retrieve the value of a semaphore's thread count (by looking at a field in the semaphore's `sem_info` structure, as described later), you should only do so for amusement—while you're debugging, for example.

You should never predicate your code on the basis of a semaphore's thread count.

Deleting a Semaphore

Every semaphore is owned by a team (the team of the thread that called **create_sem()**). When the last thread in a team dies, it takes the team's semaphores with it.

Prior to the death of a team, you can explicitly delete a semaphore through the **delete_sem()** call. Note, however, that **delete_sem()** must be called from a thread that's a member of the team that owns the semaphore—you can't delete another team's semaphores.

You're allowed to delete a semaphore even if it still has threads in its queue. However, you usually want to avoid this, so deleting a semaphore may require some thought: When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continue—they all return from **acquire_sem()** at once. You can distinguish between a "normal" acquisition and a "semaphore deleted" acquisition by the value that's returned by **acquire_sem()** (the specific return values are listed in the function descriptions, below).

Inter-application Semaphores

The **sem_id** number that identifies a semaphore is a system-wide token—the **sem_id** values that you create in your application will identify your semaphores in all other applications as well. It's possible, therefore, to broadcast the **sem_id** numbers of the semaphores that you create and so allow other applications to acquire and release them—but it's not a very good idea.

A semaphore is best controlled if it's created, acquired, released, and deleted within the same team.

If you want to provide a protected service or resource to other applications, you should accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Threads and Teams Function Summary

Function Summary

Function	Description
estimate_max_scheduling_latency()	Returns an estimate of a specified thread's scheduling latency
exit_thread()	Instructs the calling thread to exit
find_thread()	Locates the thread with a given name
get_next_team_info()	Enumerates information about all teams
get_next_thread_info()	Returns information about all threads in a given team
get_team_info()	Returns information about a given team
get_thread_info()	Returns information about a given thread
has_data()	Tests whether there is unread data in a given thread's message cache
kill_team()	Instructs a given team to exit
kill_thread()	Instructs a given thread to exit

on_exit_thread()	Sets up a callback to be executed when the calling thread exits
receive_data()	Retrieves a message from a thread's message cache
rename_thread()	Renames a thread
resume_thread()	Instructs a new or suspended thread to execute
send_data()	Places data into a thread's message cache
set_thread_priority()	Sets the a given thread's execution priority
snooze()	Pauses the calling htread for a given number of microseconds
snooze_until()	Pauses the calling thread until a specific time
spawn_thread()	Creates a new (suspended) thread
suggest_thread_priority()	Returns a suggested priority for a new thread
suspend_thread()	Halts execution of a

	given thread
wait_for_thread()	Waits for a given thread to die

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMediaAddOn Function Summary

Function Summary

Function	Description
AutoStart()	Hook: called to let nodes be instantiated on startup
CountFlavors()	Hook: return the number of flavors your add-on deals with
GetConfigurationFor()	Hook: save pertinent information about the specified node
GetFileFormatList()	Hook: return a list of file formats the node supports
GetFlavorAt()	Hook: return information about flavors supported by the add-on
ImageID()	Returns the add-on's image ID
InitCheck()	Hook: return a string explaining why the most recent error occurred
InstantiateNodeFor()	Hook: instantiate a node given information about a flavor
make_media_addon()	Hook: C function to be exported; should return a BMediaAddOn object
BMediaAddOn()	Constructor
SniffRef()	Hook: attempt to determine the MIME type of a file
SniffType()	Hook: given the MIME type of a file, identify how well your add-on can process it

SniffTypeKind()	Hook: given the MIME type of a file and some kinds, identify how well your add-on can process it
WantsAutoStart()	Hook: return true if one or more node flavors want to be instantiated when the add-on is loaded

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Archiving

The BeOS provides a protocol for *archiving* and *unarchiving* objects. When you archive an object, you record its state into a BMessage that can be sent to another application, flattened and saved as a file, cached in memory, and so on. Unarchiving does the opposite: It takes a BMessage archive and turns it into a functioning object.

The archiving protocol is represented by the BArchivable class. Many of the BeOS classes inherit from BArchivable. If you create your own classes and want them to be archivable, they, too, must inherit (directly or indirectly) from BArchivable.

The sections below tell you **(1)** how to archive and unarchive an object, and **(2)** what you have to do to create an archivable class of your own.

Archiving and Unarchiving

To archive a BArchivable object, you create a BMessage and pass it to the object's **Archive()** function. You can then send the message to some other application or, as shown below, write out its flattened form to a file so you can resurrect it later:

```
/* Archiving and storing a BButton, sans error checks.*/
BMessage message;
BButton *button;
BFile file;

button->Archive(&message);
file.SetTo(filename, B_CREATE_FILE | B_WRITE_ONLY);
message.Flatten(&file);
```

To unarchive, you find an archived object and pass it to the object's constructor, or, if you don't know the object's class (we'll talk more about that later), you pass it to **instantiate_object()**. The constructor case is simple:

```
BMessage msg;
BFile f;
BButton *button;

file.SetTo(filename, B_READ_ONLY);
msg.Unflatten(&f);
button = new BButton(&msg);
```

Invoking the constructor is fine if you know the class of the object that you're unarchiving. But consider the case where you're unarchiving an object from a message that you didn't create. For example, let's say your application displays a view that's imported from a foreign archive (sent from another application, or loaded from an add-on). To unarchive the "unknown" object, you pass the BMessage archive to **instantiate_object()**, and then cast the returned object (a BArchivable *) to some:

```
BView *foreignView;
foreignView = dynamic_cast<BView *>(instantiate_object(&msg));
```

You still don't know the class of the object, and you won't be able to invoke any functions that it defines, but at least

Each archivable class implements **Archive()** to record the object properties (data members, "owned" objects, etc.) that the class defines. These properties are added as fields to the argument BMessage. Most implementations look something like this:

```
status_t MyClass::Archive(BMessage *archive, bool deep)
{
```

```
   baseClass::Archive(archive, deep);[1]

   archive->AddInt32("MyClass::Property1", property1);[2]
   archive->AddString("MyClass::Property2", property2);
   ...

   if ( deep[3]) {
      BMessage childArchive;
      for (int32 i; i < CountChildren(); i++      )
         if ( childAt(i)->Archive(&childArchive, deep) == B_OK )
            archive->AddMessage("children", &childArchive);
   }

   message->AddString("add_on", "application/x-CodeForThisObject");[4]
}
```

[1] First, call the inherited version of **Archive()**. This gives all the ancestor classes a shot at recording the properties they define. If your object doesn't want to include shot at recording the properties they define. If your object doesn't want to include

```
BArchivable::Archive(archive, deep);
```

[2] Next, record the properties that are defined by this class. How the message fields are named and typed are up to the class itself. It's a good idea to give your fields some sort of prefix to keep them from colliding with the Be-defined fields (whose names are, unfortunately, a bit haphazard).

[3] **Archive()**'s second argument indicates whether the archive should be *deep* or *shallow*. In a deep archive, a class archives any objects that it "owns"—roughly speaking, these are the objects that the class is responsible for deleting. The archivable children must also inherit from BArchivable.

[4] This line records the signature of the library that contains the object's code. If you want your object to be loaded dynamically by some other application, you should add the library signature under the string field "add_on". You don't have to supply a library if you don't care about dynamic loading.

If a class doesn't have any data to add to the archive, it doesn't need to implement **Archive()**.

Instantiability

To be unarchivable, a class must implement a constructor that takes a BMessage archive as an argument, and it must implement the static **Instantiate()** function.

A typical unarchiving constructor calls the inherited version, and then reconstitutes the properties that were added when the object was archived:

```
```

```
MyClass::MyClass(BMessage *archive) : baseClass(archive)
{
   int32    i;
   BMessage msg;
   BArchivable *obj;

   archive->FindInt32("MyClass::Property1", &property1);
   archive->FindString("MyClass::Property2", &property2);
   ...

   while ( data->FindMessage("children", i++, &msg) == B_OK){
      obj = instantiate_object(&msg);
      childList->AddItem(dynamic_cast<ChildClass *>(obj));
   }
}
```

The class must also implement the static **Instantiate()** function (declared in BArchivable), which needn't do much more than call the archive-accepting constructor, and return a BArchivable pointer. For example:

```
BArchivable *TheClass::Instantiate(BMessage *archive)
{
   if ( validate_instantiation(archive, "TheClass"))
      return new TheClass(archive);
   return NULL;
}
```

The **validate_instantiation()** function, provided by the Support Kit, is a safety check that makes sure the BMessage object is, in fact, an archive for the named class.

Unarchiving

To unarchive a BMessage, you call the **instantiate_object()** function. When passed a BMessage archive, **instantiate_object()** looks for the first name in the "class" array, finds the **Instantiate()** function for that class, and calls it. Failing that, it picks another name from the "class" array (working up the inheritance hierarchy) and tries again.

instantiate_object() returns a BArchivable instance. You then use **cast_as()** to cast the object to a more interesting class. A typical unarchiving session looks something like this:

```
/* archive is the BMessage that we want to turn into an object.
* In this case, we want to turn it into a BView.
*/
BArchivable *unarchived = instantiate_object(archive);
if ( unarchived ) {
   BView *view = cast_as(unarchived, BView);
   if ( view ) {
      . . .
```

```
    }
}
```

Dynamic Loading

It's not defined how a host will interact with an unarchived instance of a previously unknown class. It's up to the parties to define entry points and protocols, just as it is for any other add-on module.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Debugging Tools

Declared in: be/support/Debug.h
Summary: *more...*

The Support Kit provides a set of macros that help you debug your application. These tools let you print information to standard output or to the serial port, and conditionally enter the debugger.

To enable the Support Kit's debugging macros, simply compile your code with the DEBUG preprocessor variable defined. The debugging facilities also provide a debug flag which you can use to selectively turn debugging on and off in your code. You therefore have a both compile time and run time methods for enabling debugging. Note that the DEBUG variable supercedes the runtime debug flag; if it is not defined, debugging will be disabled, irrespective of the state of the debug flag.

Serial debugging is enabled if you hold down the F1 key (Intel) or the Delete key (Macintosh) while booting into BeOS.

The DEBUG Preprocessor Variable

You can define the `DEBUG` preprocessor variable by adding the following line to your **makefile**:

```
CFLAGS := -DDEBUG
```

There is currently no mechanism for specifying #define's from the MetroWerks IDE. If you are building with the IDE, you must add the following line to your source files:

```
#define DEBUG
```

When you're through debugging your application, simply remove the `DEBUG` definition and the debugging macros will be compiled away—you don't have to actually go into the code and remove the macros or comment them out.

The Debug Flag

The `SET_DEBUG_ENABLED()` macro turns the application-wide debug flag on or off. Debug macros perform their intended functions only if `DEBUG` is #defined and the debug flag is turned on. Unlike the `DEBUG` preprocessor variable, which enables or disables debugging for sections of code, the state of the debug flag follows the flow of program execution; turning the debug flag on before a function call, for example, will cause it to be on inside the function as well.

Note: The debug flag is on by default. If you want to (initially, at least) turn it off, you should call `SET_DEBUG_ENABLED(FALSE)` as one of your first acts in `main()`.

Macros

- ASSERT() see DEBUGGER()
- ASSERT_WITH_MESSAGE() see DEBUGGER()

- DEBUG_ONLY()

DEBUG_ONLY(*expression***)**

This macro blindly executes *expression* and is useful when you wish to execute a piece of code only when debugging is enabled.

- DEBUGGER() , TRESPASS() , ASSERT() , ASSERT_WITH_MESSAGE()

DEBUGGER(*var_args***)**
TRESPASS(*void***)**
ASSERT(*condition***)**
ASSERT_WITH_MESSAGE(*condition***, *message*)**

These macros cause your program to enter the debugger: `DEBUGGER() and TRESPASS()` always enter the debugger, while `ASSERT()` and `ASSERT_WITH_MESSAGE()` enter only if *condition* (which can be any normal C or C++ expression) evaluates to **FALSE**.

`DEBUGGER()` takes a `printf()`-style variable-length argument that must be wrapped inside a second set of parentheses; for example:

```
DEBUGGER(("What time is it? %fn", system_time()));
```

The argument is evaluated and printed in the debugger's shell.

`TRESPASS()` is equivalent to `DEBUGGER("Should not be here");`.

If `ASSERT()` enters the debugger, the following message is printed:

`Assert failed: File:` *filename*`, Line:` *number*`.` *condition*

If `ASSERT_WITH_MESSAGE()` enters the debugger, the following message is printed:

`Assert failed: File:` *filename*`, Line:` *number*`.` *message*

Note that the *condition* argument needn't be wrapped in a second set of parentheses.

- IS_DEBUG_ENABLED() see SET_DEBUG_ENABLED()

- PRINT() , SERIAL_PRINT()

PRINT(*var_args*)
SERIAL_PRINT(*var_args*)

These macros print the message given by *var_args*. The argument takes the variable argument form of a `printf()` call and must be wrapped inside a second set of parenthesis; for example:

```
PRINT(("The time is %fn", system_time()));
```

`PRINT()` sends the message to standard out; `SERIAL_PRINT()` to a serial port at a data rate of 19,200 bits per second. The output is sent to **/dev/ports/serial4** on BeBoxes, **/dev/modem** on Macs, and **/dev/ports/serial1** on Intel machines.

- PRINT_OBJECT()

PRINT_OBJECT(*object*)

Prints information about the argument *object* (which must be a pointer to a C++ object) by calling the object's `PrintToStream()` function. The macro doesn't check to make sure that object actually implements the function, so you should use this macro with care.

Object information is always printed to standard out (there isn't a serial port version of the call).

- SERIAL_ see PRINT()
- SERIAL_ see TRACE()

- SET_DEBUG_ENABLED() , IS_DEBUG_ENABLED()

SET_DEBUG_ENABLED(*flag*)
IS_DEBUG_ENABLED(*void*)

The `SET_DEBUG_ENABLED()` macro sets the state of the run time debug flag: a **`TRUE`** argument turns it on, **`FALSE`** turns it off. When the flag is on, the other debugging macros work; when it's off, they're ignored. The debug flag is set to **`TRUE`** by default.

The debug flag is only meaningful if your code was compiled with the `DEBUG` preprocessor variable defined. Without this definition, the debugging macros will always be disabled.

`IS_DEBUG_ENABLED()` returns the current state of the debug flag.

TRACE() , SERIAL_TRACE()

TRACE(*void*)
SERIAL_TRACE(*void*)

These macros print the name of the source code file that contains the currently executing code (in other words, the file that contains the `TRACE()` call itself), the line number of the code, and the **`thread_id`** of the calling thread. The information is printed in this form:

```
File: filename, Line: number, Thread: id
```

`TRACE()` sends the message to standard out; `SERIAL_TRACE()` to a serial port at a data rate of 19,200 bits per second. The output is sent to **/dev/ports/serial4** on BeBoxes, **/dev/modem** on Macs, and **/dev/ports/serial1** on Intel machines.

TRESPASS() see DEBUGGER()

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Debugging Tools Function Summary

Function Summary

Function	Description
ASSERT_WITH_MESSAGE()	Enters the debugger if condition fails and debugging is enabled
DEBUGGER()	Enters the debugger if debugging is enabled
DEBUG_ONLY()	Executes code if debugging is enabled
IS_DEBUG_ENABLED()	Tests runtime debugging flag if debugging is enabled
PRINT()	Output message to standard out if debugging is enabled
PRINT_OBJECT()	Calls **`PrintToStream()`** if debugging is enabled
ASSERT()	Enters the debugger if condition fails and debugging is enabled
SERIAL_PRINT()	Output message to serial port if debugging is enabled
SERIAL_TRACE()	Prints trace to serial port if debugging is enabled
SET_DEBUG_ENABLED()	Sets runtime debugging flag if debugging is enabled
TRACE()	Prints trace to standard output if debugging is enabled
TRESPASS()	Enters the debugger if debugging is enabled

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBlockCache

Derived from: none
Declared in: be/support/BlockCache.h
Library: libbe.so
Summary: *more...*

The BBlockCache class pre-allocates a pool of fixed-length blocks of memory. You call `Get()` to retrieve a block of memory from the pool, and `Save()` to return it. BBlockCache objects are useful in applications that frequently allocate and deallocate fixed-sized objects, and in the implementations of a class' **`new`** and **`delete`** operators.

Constructor and Destructor

BBlockCache()

```
BBlockCache(size_t count, size_t size, uint32 type)
```

Creates a new memory block pool, allocating memory for *count* blocks, each controlling *size* bytes of memory. *type* is either:

- **B_OBJECT_CACHE**; memory is managed through **new** and **delete**.
- **B_MALLOC_CACHE**; memory is managed through **malloc()** and **free()**.

~BBlockCache()

```
~BBlockCache()
```

Frees any unused memory in the object's block pool. Memory that was retrieved through **Get()** (and that hasn't been returned through **Save()**) is not deallocated.

Member Functions

Get()

```
void *Get(size_t size)
```

Retrieves a block of memory of the given *size* and returns it directly. If *size* is the same as the *size* argument you passed to the constructor, the memory returned will be taken from the object's cache. Otherwise, it's allocated using either **new** or **malloc()** as requested in the constructor. When you're done with the memory, you can either deallocate it yourself, or return it to the BBlockCache object by calling **Save()**.

Save()

```
void Save(void *pointer, size_t size)
```

Returns, to the BBlockCache object, *size* bytes of memory pointed to by *pointer*. If the memory was taken from the object's pool, the memory is returned to the pool. Otherwise, it's deallocated. In either case, the caller is freed of responsibility for deallocating the memory.

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BBlockCache Function Summary

Function Summary

Function	Description
BBlockCache()	Constructor
~BBlockCache()	Destructor
Get()	Retrieves a memory block from the cache
Save()	Returns a memory block to the cache

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageRunner

Derived from: (none)
Declared in: be/app/MessageRunner.h
Library: libbe.so
Allocation: Constructor only
Summary: *more...*

The BMessageRunner class provides a handy mechanism for automatically sending an arbitrary message to a BMessenger at specified intervals. The application that creates the BMessageRunner can specify the message, the BMessenger to send the message to, how often to send the message, and how many times it should be sent.

The system roster handles actually dispatching the messages to the appropriate BMessengers at the desired time intervals; this class simply acts as an intermediary through which your application asks the roster to schedule sending the messages.

Constructor and Destructor

BMessageRunner()

```
BMessageRunner(BMessenger target, const BMessage *message,
    bigtime_t interval, int32 count = -1)
BMessageRunner(BMessenger target, const BMessage *message,
    bigtime_t interval, int32 count, BMessenger replyTo)
```

Tells the roster to send the specified *message* to the *target* BMessenger every *interval* microseconds. The message will be sent *count* times (if *count* is -1, the message will be sent forever, or until the BMessageRunner is reconfigured or deleted).

The second form of the constructor lets the application specify, in *replyTo*, the BMessenger to which replies to the message should be sent.

The BMessageRunner can be reconfigured (to change the *interval* or *count*) by calling `SetInterval()` and `SetCount()`.

After constructing a BMessageRunner, you should call `InitCheck()` to ensure that the object was created properly.

~BMessageRunner()

```
virtual ~BMessageRunner()
```

Asks the roster to stop sending the message.

Member Functions

GetInfo()

```
status_t GetInfo(bigtime_t *interval, int32 *count) const
```

`GetInfo()` returns in *interval* the time in microseconds that will pass between messages being sent, and in *count* the number of times the message will be sent.

RETURN CODES

- B_OK. Information returned successfully.
- **`B_NAME_NOT_FOUND`**. The roster returned invalid information about the BMessenger.
- **`B_BAD_VALUE`**. The roster returned invalid information about the BMessenger.
- Other errors. In general, getting an error back from this function is a bad thing.

InitCheck()

```
status_t InitCheck(void) const
```

`InitCheck()` returns a result code indicating **`B_OK`** if the BMessageRunner constructor executed sucessfully, or some other value if an error occurred setting up the object. You should call this immediately after creating a BMessageRunner, and shouldn't use the object if this function returns anything but **`B_OK`**.

SetCount() , SetInterval()

```
status_t SetCount(int32 count)
status_t SetInterval(bigtime_t interval)
```

`SetCount()` sets the number of times the BMessageRunner will send the message. If you want the message to be sent forever (until the object is deleted or `SetCount()` is called again), specify -1.

`SetInterval()` sets the number of microseconds that will pass between messages being sent.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMessageRunner Function Summary

Function Summary

Function	Description
GetInfo()	Returns the current interval and count values
InitCheck()	Returns information about the success of the constructor
BMessageRunner()	Constructor
~BMessageRunner()	Destructor
SetCount()	Sets the number of times the message will be sent
SetInterval()	Sets the delay between message transmissions

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStopWatch

Derived from: none
Declared in: be/support/StopWatch.h
Library: libbe.so
Summary: *more...*

The BStopWatch class is a debugging tool designed to time the execution of portions of your code. When a BStopWatch object is constructed, it starts an internal timer. When it's deleted, it stops the timer and prints the elapsed time to standard out in this format:

- StopWatch "*name*": *f* usecs.

Where *name* is the name that you gave to the object when you constructed it, and *f* is the elapsed time in microseconds.

Look at all these other things you can do...

- Suspend, resume, and reset the timer (`Suspend()`, `Resume()`, `Reset()`).
- Ask for the current elapsed time without stopping the timer (`ElapsedTime()`).
- Record "lap points" that are printed out at the end of the run (`Lap()`).

Using a BStopWatch is simple; this...

```
BStopWatch *watch = new BStopWatch("Timer 0");
/* The code you want to time goes here. */
delete watch;
...
```

...will produce, on standard out, a message that goes something like this:

```
StopWatch "Timer 0": 492416 usecs.
```

This would indicate that the timed code took about half a second to execute—remember, you're looking at microseconds.

If you want to time an entire function, just toss a StopWatch on the stack:

```
void MyFunc()
{
   BStopWatch watch("Timer 0");
   ...
}
```

When the function returns, the BStopWatch prints its message.

BStopWatch objects are useful if you want to get an idea of where your cycles are going. But you shouldn't rely on them for painfully accurate measurements.

Constructor and Destructor

BStopWatch()

```
BStopWatch(const char *name, bool silent = false)
```

Creates a BStopWatch object, names it, and starts its timer. If *silent* is **false** (the default), the object will print its elapsed time when it's destroyed; otherwise the message isn't printed. To get the elapsed time from a silent BStopWatch, call **ElapsedTime()**.

~BStopWatch()

```
~BStopWatch()
```

Stops the object's timer, prints a timing message to standard out (unless it's running silently), and then destroys the object. By default the timing message looks like this:

- StopWatch "*name*": *f* usecs.

If you've recorded some lap points (through the **Lap()** function), you'll also see the lap times as well:

- StopWatch "*name*": *f* usecs.
  [*lap#*: *soFar#thisLap*] [*lap#*: *soFar#thisLap*] [*lap#*: *soFar#thisLap*]...

...where *lap#* is the number of the lap, *soFar* was the total elapsed time at that lap, and *thisLap* was the time it took to complete the lap.

Member Functions

ElapsedTime()

```
bigtime_t ElapsedTime(void) const
```

Returns the elapsed time, in microseconds, since the object was created or last **Reset()**. This function doesn't print the time message, nor does it touch the timer (the timer keeps running—unless it's paused).

```
BStopWatch watch("Timer 0");
...
printf("Elapsed time: %Ldn", watch.ElapsedTime());
```

Lap()

```
bigtime_t Lap()
```

Records a "lap point" and returns the total elapsed time so far. The lap point times are printed individually when the object is destroyed, provided the object isn't silent. You can record as many as eight lap points; if you ask for a ninth lap point, the lap isn't recorded and this function returns 0. See **~BStopWatch()** for a description of what the lap points look like when they're printed.

Name()

```
const char *Name(void) const
```

Returns the name of the object, as set in the constructor.

Reset() see Suspend()

Resume() see Suspend()

Suspend() , Resume() , Reset()

```
void Suspend(void)
void Resume(void)
void Reset(void)
```

These functions affect the object's timer.

`Suspend()` stops the timer but doesn't reset it.

`Resume()` starts the timer running again.

`Reset()` sets the elapsed time to 0, but doesn't stop the timer. You can call `Reset()` at any time, regardless of whether the object is running or suspended.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BStopWatch Function Summary

Function Summary

Function	Description
ElapsedTime()	Returns time elapsed since the last `Reset()`
Lap()	Records a lap point
Name()	Returns the name of the object
Reset()	Resets the timer
Resume()	Resumes the timer
BStopWatch()	Constructor
~BStopWatch()	Destructor
Suspend()	Suspends the timer

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Support Kit

The Support Kit contains classes and utilities that any application can take advantage of. Among other things, it includes:

- The BArchivable protocol for objects that can archive themselves and be instantiated from their archives.
- The BBlockCache class provides a simple mechanism for caching fixed-length blocks of memory.
- The BList class, a container for keeping track of a number of data items, typically object pointers.
- The BLocker and BAutolock classes for implementing locking mechanisms.
- The BDataIO and BPositionIO protocols for objects that can be read and written, and the BMemoryIO and BMallocIO classes that implement the protocols.
- The BFlattenable protocol for objects that can write themselves to a flattened representation.
- The BMessageRunner class sends a message over and over.
- The BString class is an interface to smart string allocation, storage, and manipulation. Call me weak, but I can't live without it.

- The BStopWatch class times your functions. Use it when you're developing your app.
- Common defined types and constants, such as **int32** and **int64**.
- The error codes for (theoretically) all the software kits. (Some error code may have snuck into other kits.)
- Type codes for identifying data types in messages, attributes, and resources.
- Utility functions such as **atomic_add()**, **write_16_swap()**, and **convert_to_utf8()**.
- "Thread Local Storage" functions that let you define global variables whose values depend on the thread from which they're accessed. These functions are handy when you're porting code that was designed for a single-threaded system.

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Thread Local Storage

Declared in: be/support/TLS.h
Library: libbe.so
Summary: *more...*

The Thread Local Storage (TLS) functions let you create global variables that refer to different data depending on the thread from which the variables are referenced. In essence, TLS variables let you extend the data that's associated with a given thread. This can be particularly important when you're porting code that wasn't designed for a multi-threaded system.

To create a TLS variable, you call **tls_allocate()**:

```
int32 myName = tls_allocate();
int32 myStatus = tls_allocate();
```

You only allocate a given TLS variable once (i.e. you don't allocate the same variable in each thread). Typically, you would call **tls_allocate()** from an app-initialization routine.

To set and retrieve the value of a TLS variable, you call **tls_set()** and **tls_get()**:

```
void SetName(const char *name) {
   tls_set(myName, (void *)name);
}

void SetStatus(int32 state) {
   tls_set(myStatus, (void *)(int32 *)&state);
}
void Report() {
   if (tls_get(myStatus) != B_OK) {
      printf("Error in %sn", tls_get(myName);
}
```

The values that **tls_set()** and **tls_get()** set and return are thread-specific; in the examples above, each thread has its own values for **myName** and **myStatus**. Note that **tls_set()** and **tls_get()** operate in the context of the calling thread only—you can't set or retrieve a TLS value for some other thread.

You never access a TLS variable (i.e. the value returned by **tls_allocate()**) directly. You may only use it as an argument to the other TLS functions.

Thread Local Storage Functions

tls_allocate()

```
int32 tls_allocate(void)
```

Creates and returns a new TLS variable. Each thread in your application will have its own copy of this variable; setting the value of the variable (through `tls_set()`) in one thread won't affect its value in any other thread. You should only allocate a TLS variable once per team—you don't have to allocate it in each thread.

The memory for the variable's data isn't allocated by `tls_allocate()`; see `tls_set()` for more information on TLS memory allocation).

tls_address()

```
void *tls_address(int32 tlsVariable)
```

Returns the address of the memory that holds the value that's referred to by *tlsVariable*. The latter is a TLS variable that must have been created through `tls_allocate()`. The value returned is valid for the calling thread only.

tls_set() , tls_get()

```
void tls_set(int32 tlsVariable, void *data)
void *tls_get(int32 tlsVariable)
```

`tls_sets()` sets *tlsVariable* to refer to *data* within the context of the calling thread. *data*, which must already be allocated, can point to data of any size or type. The data needn't be the same type or size for each thread; for example, a given TLS variable *hodgePodge* can refer to a string in one thread, an integer in another, an object in another, and so on.

`tls_get()` returns a pointer to the data referred to by *tlsVariable* within the context of the calling thread. If *tlsVariable* hasn't been allocated (`tls_allocate()`), or if its value hasn't been set (`tls_set()`), the function returns `NULL`.

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Thread Local Storage Function Summary

Function Summary

Function	Description
tls_address()	Returns the address of the data behind a TLS variable
tls_allocate()	Creates a new TLS variable
tls_set()	Sets and retrieves the value of a TLS variable

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BArchivable Function Summary

Function Summary

Function	Description
BArchivable()	Constructor
~BArchivable()	Destructor
Archive()	Archives a BArchivable into a BMessage
Instantiate()	Creates a BArchivable from a BMessage

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BHandler Function Summary

Function Summary

Function	Description
AddFilter()	Adds a filter to the BHandler's list of filters
Archive()	Archives a BHandler into a BMessage
FilterList()	Returns the list of filters for the BHandler
GetSupportedSuites()	Hook called in response to **`B_GET_SUPPORTED_SUITES`** scripting requests
BHandler()	Constructor
~BHandler()	Destructor
LockLooper()	Locks the BLooper that the BHandler is associated with
LockLooperWithTimeout()	Locks, with a timeout, the BLooper that the BHandler is associated with
Looper()	Returns the BLooper that the BHandler is associated with
MessageReceived()	Hook called by the BHandler to respond to messages
Name()	Returns the name identifying the BHandler
NextHandler()	Gets the next BHandler
RemoveFilter()	Removes a filter from the

	BHandler's list of filters
ResolveSpecifier()	Hook called to respond to scripting requests
SetFilterList()	Sets the list of filters for the BHandler
SetName()	Sets the name identifying the BHandler
SetNextHandler()	Sets the next BHandler
StartWatchingAll()	Begin watching for notices from this handler
StartWatching()	Begin watching for a specific notice from this handler
StopWatchingAll()	Stop watching for notices from this handler
StopWatching()	Stop watching for a specific notice from this handler
UnlockLooper()	Unlocks the BLooper that the BHandler is associated with

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BLooper Function Summary

Function Summary

Function	Description
AddCommonFilterList()	Adds a filter applicable to all messages received by the BLooper
AddHandler()	Adds a BHandler to the BLooper
CommonFilterList()	Returns the list of filters applicable to all messages received by the BLooper
CountHandlers()	Returns the number of BHandlers attached to the BLooper
CountLockRequests()	Returns the number of threads currently trying to lock the BLooper
CountLocks()	Returns the number of times the locking thread has locked the BLooper
CurrentMessage()	Returns the message currently being processed by the BLooper
DetachCurrentMessage()	Removes the current message from the message queue
DispatchMessage()	Hook called to dispatch messages received by the BLooper

HandlerAt()	Returns the BHandler at a given index in the BHandler list
IndexOf()	Returns the index of a given BHandler in the BHandler list
IsLocked()	Tests if the calling thread holds the BLooper lock
Lock()	Locks the BLooper
LockWithTimeout()	Locks the thread if possible within a given time period
LockingThread()	Returns the thread_id of the thread holding the BLooper lock
BLooper()	Constructor
~BLooper()	Destructor
LooperForThread()	Returns the BLooper that spawned a given thread
MessageQueue()	Returns the BMessageQueue for the object
MessageReceived()	Provided for implementation
PostMessage()	Delivers a BMessage to the BLooper
PreferredHandler()	Returns the BLooper's preferred handler
Quit()	Shuts down and deletes the BLooper
QuitRequested()	Hook called in response to a **B_QUIT_REQUESTED** message
RemoveCommonFilterList()	Removes a filter applicable to

	all messages received by the BLooper
RemoveHandler()	Removes a BHandler from the BLooper
Run()	Spawns and runs the message loop thread
Sem()	Returns the semaphore used by the object for locking
SetCommonFilterList()	Sets the list of filters applicable to all messages received by the BLooper
SetPreferredHandler()	Sets the object's preferred handler
Team()	Returns the team of the message loop
Thread()	Returns the thread of the message loop
Unlock()	Unlocks the BLooper

The Be Book **The Application Kit** **The Application Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Application Kit

The Application Kit is the starting point for all applications. Its classes establish an application as an identifiable entity—one that can cooperate and communicate with other applications.

The Application Kit is divided into these topics:

- BApplication. All but the simplest of applications must have one (and only one) BApplication object. This object is typically an instance of a BApplication subclass that you create especially for your application. The BApplication object makes a connection to the App Server and runs the application's main message loop.
- Messaging. The kit provides a messaging service that lets threads talk to each other. This service can deliver messages within your own application, or from one application to another. It's also used by the system to deliver user event messages (key clicks, mouse moves) to your application. Most of the Application Kit's classes are involved in the messaging system.
- Scripting. The objects that you create can be controlled by commands issued from other applications.
- BRoster. The BRoster object keeps track of all running applications. It can identify applications, launch them, and provide the information needed to set up communications with them.

- BClipboard. The BClipboard object provides an interface to the clipboard where cut and copied data can be stored, and from which it can be pasted.

- BCursor. You use BCursor objects to represent distinct cursors. Functions defined by BApplication and BView let you assign your cursors to your entire application, or to individual views .

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BClipboard

Derived from: *none*
Declared in: be/app/Clipboard.h
Library: libbe.so
Allocation: Constructor, on the stack, or use the **be_clipboard** global object.
Summary: *more...*

A BClipboard object is an interface to a *clipboard*, a resource that provides system-wide, temporary data storage. Clipboards are identified by name; if two apps want to refer to the same clipboard, they simply create respective BClipboard objects with the same name:

```
/* App A: This creates a clipboard named "MyClipboard". */
BClipboard *appAclipboard = new BClipboard("MyClipboard");

/* App B: This object refers to the clipboard already created by App A. */
BClipboard *appBclipboard = new BClipboard("MyClipboard");
```

The System Clipboard

In practice, you rarely need to construct your own BClipboard object; instead, you use the BClipboard that's created for you by your BApplication object. This object, which you refer to through the global **be_clipboard** variable, accesses the default system clipboard. Data that you write to your **be_clipboard** object can be read from any other app's **be_clipboard**. For example, the cut/copy/paste operations defined by BTextView transfer data through the system clipboard.

To access the system clipboard without creating a BApplication object, construct a BClipboard object with the name "system". The system clipboard is under the control of the user—you should only read or write the system clipboard as a direct result of the user's actions. If you create your own clipboards don't name them "system".

The Clipboard Message

To access a clipboard's data, you call functions on a BMessage that the BClipboard object hands you (through its **Data()** function). The BMessage follows these conventions:

- The **what** value is unused.
- The data is stored in a message field. The field should be typed as **B_MIME_TYPE**; the MIME type that describes the data should be used as the name of the field that holds the data (see "Writing to the Clipboard" for an example).
- If the BMessage contains more than one field, each field should present the same data in a different format. For example, the **StyledEdit** app writes text data in its own format (in

order to encode the fonts, colors, etc.) and also writes the data as plain ASCII text (MIME type "text/plain").

Writing to the Clipboard

The following annotated example shows how to write to the clipboard.

```
BMessage *clip = (BMessage *)NULL;

if (be_clipboard->Lock()[1]) {
   be_clipboard->Clear()[2];
   if ((clip = be_clipboard->Data()[3]) {
      clip->AddData("text/MyFormat", B_MIME_TYPE, myText, myLength)[4];
      clip->AddData("text/plain", B_MIME_TYPE, asciiText, asciiLength)[4];
      be_clipboard->Commit()[5];
   }
   be_clipboard->Unlock()[6];
}
```

1 **Lock()** your BClipboard object. This uploads data from the clipboard into your BClipboard's local BMessage object, and prevents other threads in your application from accessing the BClipboard's data. Note that locking does *not* lock the underlying clipboard data—other applications can change the clipboard while you have your object locked.

2 Prepare the BClipboard for writing by calling **Clear()**. This erases the data that was uploaded from the clipboard.

3 Call **Data()** to get a pointer to the BClipboard's BMessage object.

4 Write the data by invoking **AddData()** directly on the BMessage. In the example, we write the data in two different formats.

5 Call **Commit()** to copy your BMessage back to the clipboard. As soon as you call **Commit()**, the data that you added is visible to other clipboard clients.

6 **Unlock()** balances the **Lock()**. The BClipboard object can now be accessed by other threads in your application.

If you decide that you don't want to commit your changes, you should call **Revert()** before you unlock.

Reading from the Clipboard

Here we show how to read a simple string from the clipboard.

```
const char *text;
int32 textLen;
BMessage *clip = (BMessage *)NULL;
```

```
if (be_clipboard->Lock()[1]) {
   if ((clip = be_clipboard->Data();
      clip->FindData("text/plain", B_MIME_TYPE,
         (const void **)&text, &textlen)[2];
   be_clipboard->Unlock()[3];
}
```

[1] As in writing, we bracket the operation with `Lock()` and `Unlock()`. Keep in mind that `Lock()` uploads data from the clipboard into our object. Any changes that are made to the clipboard (by some other application) after `Lock()` is called won't be seen here.

[2] In this example, we only look for one hard-coded format. In a real application, you may have a list of formats that you can look for.

[3] It isn't necessary to examine the clipboard data *before* you unlock it. The `FindData()` call could just as well have been performed after the `Unlock()` call.

Persistence

Inter-boot persistence: Clipboard data does *not* persist between boots—the constructor provides a persistence flag, but it's currently unused.

Intra-boot persistence: Once you've created a clipboard, that clipboard will exist until you reboot your computer. For example, let's say you design an app that creates a clipboard called "MyClip": You launch the app, write something to "MyClip", and then quit the app. The clipboard—and the data that you wrote to it—will still exist: If you relaunch your app (or any app that knows about "MyClip"), you can pick up the data by reading from the "MyClip" clipboard.

Constructor and Destructor

BClipboard()

```
BClipboard(const char *name, bool discard = false)
```

Creates a new BClipboard object that refers to the *name* clipboard. The clipboard itself is created if a clipboard of that name doesn't already exist.

The *discard* flag is currently unused.

~BClipboard()

```
virtual ~BClipboard()
```

Destroys the BClipboard object. The clipboard itself and the data it contains are not affected by the object's destruction.

Member Functions

Clear() , Commit() , Revert()

status_t **Clear**(void)
status_t **Commit**(void)
status_t **Revert**(void)

These functions are used when you're writing data to the clipboard. **Clear()** prepares your BClipboard for writing. You call **Clear()** just before you add new data to your clipboard message. **Commit()** copies your BClipboard data back to the clipboard. See "Writing to the Clipboard" for an example of these functions.

Revert() refreshes the BClipboard's data message by uploading it from the clipboard. The function is provided for the (rare) case where you alter your BClipboard's data message, and then decide to back out of the change. In this case, you should call **Revert()** (rather than **Commit()**). If you don't revert, your BClipboard's message will still contain your unwanted change, even if you unlock and then re-lock the object.

All three functions returns **B_ERROR** if the BClipboard isn't locked, and **B_OK** otherwise.

Commit() see Clear()

Data()

BMessage ***Data**(void) const

Returns the BMessage object that holds the BClipboard's data, or **NULL** if the BClipboard isn't locked. You're expected to read and write the BMessage directly; however, you may *not* free it or dispatch it like a normal BMessage. If you change the BMessage and want to write it back to the clipboard, you have to call **Commit()** after you make the change.

See "The Clipboard Message" for more information.

DataSource()

BMessenger **DataSource**(void) const

Returns a BMessenger that targets the BApplication object of the application that last committed data to the clipboard. The BClipboard needn't be locked.

LocalCount() , SystemCount()

uint32 **LocalCount**(void) const
uint32 **SystemCount**(void) const

These functions return the clipboard count. **LocalCount()** uses a cached count, while **SystemCount()** asks the Application Server for the more accurate system counter.

Lock() , Unlock() , IsLocked()

bool **Lock**(void)
void **Unlock**(void)
bool **IsLocked**(void)

Lock() uploads data from the clipboard into your BClipboard object, and locks the object so no other thread in your application can use it. You must call **Lock()** before reading or writing the BClipboard. **Lock()** blocks if the object is already locked. It returns **true** if the lock was acquired, and **false** if the BClipboard object was deleted while **Lock()** was blocked.

There's no way to tell **Lock()** to time out.

Unlock() unlocks the object so other threads in your application can use it.

IsLocked() hardly needs to be documented.

Name()

const char ***Name**(void) const

Returns the name of the clipboard. The object needn't be locked.

Revert() see Clear()

StartWatching() , StopWatching()

status_t **StartWatching**(BMessenger *target*)
status_t **StopWatching**(BMessenger *target*)

If you want to be alerted when the clipboard changes, call **StartWatching()**, passing a BMessenger to be the target for the notification. When the clipboard changes, a **B_CLIPBOARD_CHANGED** message will be sent to the target.

StopWatching() stops monitoring the clipboard for changes.

RETURN CODES

- B_OK. No error.
- Other errors. You get the idea.

- StopWatching() see StartWatching()
- LocalCount()
- Unlock() see Lock()

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BClipboard Function Summary

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Mail Kit Table of Contents

- The Mail Kit
- The Mail Daemon
- BMailMessage

The Be Book **The Mail Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Mail Kit: Master Index

A

AddContent()	BMailMessage
AddEnclosure()	BMailMessage
AddHeaderField()	BMailMessage

C

check_for_mail()	The Mail Daemon
Constructing a Mail Message	BMailMessage
Constructor and Destructor	BMailMessage
count_pop_accounts()	The Mail Daemon

D

decode_base64()	The Mail Daemon

E

The E-mail File Type	The Mail Kit
encode_base64()	The Mail Daemon

F

forward_mail()	The Mail Daemon
Functions	The Mail Daemon

G

get_mail_notification()	The Mail Daemon
get_pop_account()	The Mail Daemon
get_smtp_host()	The Mail Daemon

M

The Mail Daemon and the Mail Server	The Mail Daemon
The Mail Daemon	The Mail Daemon
The Mail Daemon	The Mail Daemon
The Mail Kit	The Mail Kit
The Mail Kit	The Mail Kit
BMailMessage	BMailMessage
BMailMessage()	BMailMessage
~BMailMessage()	BMailMessage
Mail Message Files	The Mail Kit
Member Functions	BMailMessage

O

Other Mail Daemon Features	The Mail Daemon

Q

Queries and Mail Messages	The Mail Kit
Querying Mail Messages	The Mail Kit

S

Send()	BMailMessage
send_queued_mail()	The Mail Daemon
Sending and Retrieving Mail	The Mail Daemon
set_mail_notification()	The Mail Daemon
set_pop_account()	The Mail Daemon
set_smtp_host()	The Mail Daemon

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMailMessage

Derived from: none
Declared in: be/mail/E-mail.h
Library: libmail.so
Summary: *more...*

The BMailMessage class provides an easy way to send e-mail messages. If you want to do it the hard way, look up the SMTP RFC and start plodding your way through the Network Kit documentation. You'll get it working one of these days.

Or you can sail right on through and be sending e-mail from your own applications in a matter of minutes using your friend, the BMailMessage.

Constructing a Mail Message

To send an e-mail, you simply construct a new BMailMessage object, add a "To" header field, add the content, and send the message on its way. For example:

```
BMailMessage *mail;
char *message;

mail = new BMailMessage();
mail->AddHeaderField(B_MAIL_TO, "bob@uncle.com");
mail->AddHeaderField(B_MAIL_SUBJECT, "Hi");
message = "Hi, Uncle Bob!";
mail->AddContent(message, strlen(message));
```

This is a pretty basic message. The subject is "Hi," the message is sent to "bob@uncle.com," and the message body is "Hi, Uncle Bob!"

You can add other fields, including carbon-copy (CC) and blind-carbon-copy (BCC) fields, and you can add attachments. For example, if you want to also attach a file called "/boot/home/file.zip," you can do the following:

```
mail->AddEnclosure("/boot/home/file.zip");
```

Once your message has been constructed, you can send it by calling **Send()**:

```
mail->Send();
```

That's the basic technique behind sending e-mail under the BeOS. The mail daemon also fetches incoming mail from a POP server, but you can't use the BMailMessage class to read these messages; you use the BeOS BQuery and BNode classes to locate messages of interest and obtain information about them. See "Querying Mail Messages" for more information.

Constructor and Destructor

BMailMessage()

```
BMailMessage(void)
```

Creates and returns a new BMailMessage object, which is empty. You need to call other functions defined by this class to fill out the message.

~BMailMessage()

```
~BMailMessage()
```

Destroys the BMailMessage, even if the object's fields are "dirty." For example, if you create a new BMailMessage object with the intention of sending a message, fill out some or all of the fields, and then delete the object, the object is destroyed without being sent.

Member Functions

AddContent()

```
status_t AddContent(const char *text,
    int32 length,
    uint32 encoding = B_ISO1_CONVERSION,
    bool replace = false)
status_t AddContent(const char *text,
    int32 length,
    const char *encoding,
    bool replace = false)
```

Adds the specified *text* (which contains *length* characters) to the BMailMessage object's content. The text's encoding is specified by the *encoding* parameter, either directly or by pointer.

If *replace* is **true**, any existing text is deleted before the new content is added; otherwise, the specified text is appended to the end of the existing message content.

RETURN CODES

- B_OK. The content was changed without error.
- B_ERROR. Unable to add the new content.

AddEnclosure()

```
status_t AddEnclosure(entry_ref *enclosure_ref, bool replace = false)
status_t AddEnclosure(const char *path, bool replace = false)
status_t AddEnclosure(const char *mime_type,
    void *data,
    int32 length,
    bool replace = false)
```

Adds an attachment to the message. The first two forms of AddEnclosure() add a file to the message, given either an entry_ref pointer or a pathname. The third form adds a block of memory (of the given *length*) to the message as an enclosure, with the specified MIME type.

If *replace* is **true**, any existing attachments—including the body of the message—are removed before the new one is added; otherwise the new enclosure is added, leaving previous attachments intact.

If you specify **true** for *replace*, not only will all existing enclosures be discarded, but so will the content of the message body itself.

RETURN CODES

- `B_OK`. The content was changed without error.
- `B_ERROR`. Unable to add the new enclosure.

AddHeaderField()

```
status_t AddHeaderField(const char *field_name,
    const char *field_str,
    bool replace = false)
```

Adds a header field to the BMailMessage object. The value of the field whose name is specified by *field_name* is set to the string specified by *field_str*.

If *replace* is **true**, all existing header fields of the specified name are deleted before adding the new header field; if *replace* is **false**, a new header whose field is named *field_name* is added.

RETURN CODES

- `B_OK`. The content was changed without error.
- `B_ERROR`. Unable to add the new header field.

Send()

```
status_t Send(bool send_now = false, bool remove_when_sent = false)
```

Queues the message for transmission. If *send_now* is **true**, the message is sent immediately; otherwise, it is placed in the queue to be sent the next time `check_for_mail()` is called or the mail daemon performs an automatic mail check.

If the *remove_when_sent* argument is **true**, the message will be deleted from the user's disk drive after it has been sent; otherwise, it will be saved for posterity.

RETURN CODES

- `B_OK`. The content was changed without error.
- `B_MAIL_NO_RECIPIENT`. There needs to be either a "To" or "Bcc" field in the message.
- Errors from `BMessage::SendMessage()`.

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BMailMessage Function Summary

Function Summary

Function	Description
AddContent()	Add text to the message
AddEnclosure()	Add an attachment to the message
AddHeaderField()	Add a header field to the mail message
BMailMessage()	Constructor
~BMailMessage()	Destructor
Send()	Queues the message for transmission

The Be Book **The Mail Kit** **The Mail Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Mail Daemon

Derived from: none
Declared in: be/kit/mail/E-mail.h
Library: libmail.so
Summary: *more...*

Every computer running the BeOS has a **mail daemon**; this is a local process that's responsible for sending e-mail to and receiving e-mail from a mail server. The mail server that the daemon talks to is a networking application that's either part of your Internet Service Provider's services, or that's running on a local "mail repository" machine.

The functions described in this section tell you how to manage the mail daemon's connection with the mail server—how to tell the daemon which mail server to communicate with, how to tell the mail daemon to send and retrieve e-mail, how to automate mail retrieval, and so forth.

Many of the functions described here are user-accessible through the E-mail preference application. These functions should generally not be used; the settings they control belong to the user, and your application should usually avoid changing the user's settings. The only legitimate reason to use these configuration setting functions is if you want to build your own E-mail preference application.

The other functions, such as `forward_mail()`, `check_for_mail()`, `encode_base64()`, and `decode_base64()`, might be legitimately used by your e-mail program.

The architecture of an e-mail message isn't discussed here; for that information, see "Mail Messages (BMailMessage)."

The Mail Daemon and the Mail Server

The mail daemon can talk to two different kinds of mail server:

- The **Post Office Protocol** ("POP") server manages individual mail accounts. When the BeOS mail daemon wants to retrieve mail that's been sent to a user, it must tell the mail server which POP accouant it's retrieving mail for.
- The **Simple Mail Transfer Protocol** ("SMTP") server manages mail that's being sent out onto the network. Messages sent through an SMTP server will eventually find their way to a POP server to be received by the destination user.

The POP and the SMTP servers are identified by their hosts' names (in other words, the names of the machines on which the servers are running). The mail daemon can only talk to one POP and one SMTP server at a time, but can talk to the two of them simultaneously. Usually—but not always—the POP and SMTP servers reside on the same machine, and so are identified by the same name.

To set the identities of the POP host, you fill in the fields of a **mail_pop_account** structure and pass the structure to the **set_pop_account()** function. As the name of the structure implies, **mail_pop_account** encodes more than just the names of the server's host. It also identifies a specific user's POP mail account; the complete definition of the structure is this:

```
typedef struct
{
   char pop_name[B_MAX_USER_NAME_LENGTH];
   char pop_password[B_MAX_USER_NAME_LENGTH];
   char pop_host[B_MAX_HOST_NAME_LENGTH];
   char real_name[128];
   char reply_to[128];
   int32 days;
   int32 interval;
   int32 begin_time;
   int32 end_time;
} mail_pop_account;
```

The pop_name, **pop_password**, and pop_host fields in the mail_pop_account structure represent the username, password, and POP server host of the e-mail user. The real_name is the user's real name, and reply_to is the e-mail address to which replies should be sent.

The days field can contain any of the following flags to specify which days of the week the mail daemon should automatically check mail for the described account:

Constant	Meaning
B_CHECK_NEVER	Don't automatically check the account's mail.
B_CHECK_WEEKDAYS	Check the mail only on weekdays.
B_CHECK_DAILY	Check the mail every day.
B_CHECK_CONTINUOUSLY	Check continuously every day.

The interval specifies how many seconds apart each e-mail retrieval should be, and the begin_time and end_time specify the time of day (in seconds) that automatic retrieval should begin and end.

The SMTP server can be selected by calling **set_smtp_host()**, passing in a pointer to the SMTP host's name.

Sending and Retrieving Mail

Messages that are retrieved (from the mail server) by the mail daemon are stored as individual files on the user's hard disk, from whence they are plucked and displayed by a mail-reading application (a "mail reader"; Be supplies a simple mail reader called BeMail). Similarly, messages that the user composes (in a mail composition application) and sends are stored as individual files until the mail daemon comes along and passes them on to the mail server.

Sending and retrieving mail is the mail daemon's most important function. Both actions (server-to-database and database-to-server transmission) are performed through the

`check_for_mail()` function.

The BMailMessage class provides a convenient means for creating and sending new mail messages; visit the section on that class for further information and a simple example.

Mail that has been retrieved by the mail daemon can be identified and queried using the mail attributes defined by the Mail Kit. By using the BQuery class, you can scan all newly-received mail messages and parse the message file to present each message to the user. For a more in-depth discussion of the mail attributes and how to use them to your benefit, read "Querying Mail Messages."

Other Mail Daemon Features

The other mail structures and functions define the other features that are provided by the mail daemon. These features are:

- **Mail notification**. The `mail_notification` structure (passed through the `set_mail_notification()` function) lets you tell the daemon how you would like it to tap you on the shoulder when it has new mail for you to read. Would you like it to display an alert panel? Squawk at you? Both? This can be configured by the user in the E-mail preference application.
- **Mail forwarding**. The `forward_mail()` function lets you ask the Mail Kit to forward a message to one or more other accounts.
- **Base-64 encoding and decoding**. The `encode_base64()` and `decode_base64()` functions let you easily handle ASCII-encoded file attachments.

Functions

check_for_mail()

status_t **check_for_mail(**int32 **incoming_count* = NULL**)**

Sends and retrieves mail. More specifically, this function asks the mail daemon to retrieve incoming messages from the POP server and send any queued outgoing messages to the SMTP server. The number of POP messages that were retrieved are stored in the variable pointed to by *incoming_count*. If you specify **NULL** for *incoming_count*, `check_for_mail()` won't return the number of messages retrieved. You should specify **NULL** unless you really want to know how many messages were retrieved, since requesting this information could potentially slow down the retrieval process.

If all is well in the mail world, this function returns **B_OK**; otherwise, it returns a highly useful result code.

RETURN CODES

- **B_OK**. Mail was sent and retrieved without incident.
- **B_MAIL_NO_DAEMON**. The mail daemon isn't running.
- **B_MAIL_UNKNOWN_USER**. The POP server doesn't recognize the user name.
- **B_MAIL_WRONG_PASSWORD**. The POP server doesn't recognize the password.
- **B_MAIL_UNKNOWN_HOST**. The POP or SMTP server can't be found.
- **B_MAIL_ACCESS_ERROR**. The connection to the POP or SMTP server failed.

count_pop_accounts()

int32 **count_pop_accounts(**void**)**

Returns the number of POP accounts that have been configured.

The mail daemon currently supports only one POP account, so this function will always return 1. You shouldn't assume there will only be one POP account, though, as this will probably change in the future.

decode_base64()

ssize_t **decode_base64(**char **out*, char **in*, off_t *length*, bool *replace_cr* = *false***)**

Decodes the base-64 data pointed to by *in*, which is *length* bytes long, and writes the decoded output into the buffer pointed to by *out*. If *replace_cr* is **true**, carriage return characters in the output are converted into newlines, otherwise the data is returned in its original, unaltered, form.

You would typically specify *replace_cr* as **true** if you're decoding an ASCII text document, and as **false** if decoding a binary file.

This function returns the size of the output data that's been stored in the *out* buffer.

You must be certain, in advance, that the output buffer is large enough to hold the decoded data, or this function will do bad things.

---

encode_base64()

ssize_t **encode_base64**(char **out*, char **in*, off_t *length*)

Encodes the data pointed to by *in*, which is *length* bytes long, and writes the base-64 encoded output into the buffer pointed to by *out*.

This function returns the size of the output data that's been stored in the *out* buffer.

You must be certain, in advance, that the output buffer is large enough to hold the encoded data, or this function will do bad things.

---

forward_mail()

status_t **forward_mail**(entry_ref **message_ref*, const char **recipients*,
bool *now* = *true*)

Forwards the mail message specified by *message_ref* to the list of users given by *recipients*. The list of user names specified in *recipients* must be separated by commas and/or whitespace, and must be null-terminated.

If the *now* parameter is **true**, the messages will be sent immediately; if **false**, the message will be queued up to be sent the next time `check_for_mail()` is called, or the next time the mail daemon performs an automatic mail check.

RETURN CODES

- `B_OK`. The message was forwarded without error.
- `B_MAIL_NO_RECIPIENT`. No valid recipients were specified.
- Errors returned by `send_queued_mail()`, if *now* is **true**.

---

get_mail_notification() , set_mail_notification()

status_t **get_mail_notification**(mail_notification *_notification_settings_)
status_t **set_mail_notification**(mail_notification *_notification_settings_, bool _save_ = _true_)

`get_mail_notification()` fills the specified **`mail_notification`** structure with information describing how the user is currently being notified of received e-mail. There are two possible notification signals: the mail alert panel and the system beep. The **`mail_notification`** structure looks like this:

```
typedef struct
{
   bool alert;
   bool beep;
} mail_notification;
```

`get_mail_notification()` always returns **`B_OK`**. If the current settings can't be checked (for example, if the user has never configured mail), *alert* will be returned as the default value of **`false`**, and *beep* will be **`true`**.

`set_mail_notification()` accepts a pointer to a **`mail_notification`** structure and configures the system to report incoming mail using the methods specified therein. If the *save* argument is **`true`**, the change is set as the new default and will be remembered when the computer is shut down. If **`false`**, the change is temporary.

RETURN CODES

- **`B_OK`**. The notification was successfully set or retrieved.
- **`B_NO_REPLY`**. The mail daemon didn't respond to the request.

---

get_pop_account() , set_pop_account()

status_t **get_pop_account**(mail_pop_account *_account_info_, int32 _index_ = _0_)
status_t **set_pop_account**(mail_pop_account *_account_info_, int32 _index_ = _0_)

Get and set the specified POP account's information. The **`mail_pop_account`** structure is defined as follows:

```
typedef struct
{
   char pop_name[B_MAX_USER_NAME_LENGTH];
   char pop_password[B_MAX_USER_NAME_LENGTH];
   char pop_host[B_MAX_HOST_NAME_LENGTH];
   char real_name[128];
```

```
    char reply_to[128];
    int32 days;
    int32 interval;
    int32 begin_time;
    int32 end_time;
} mail_pop_account;
```

The `pop_name`, **`pop_password`**, and `pop_host` fields in the `mail_pop_account` structure represent the username, password, and POP server host of the e-mail user. The `real_name` is the user's real name, and `reply_to` is the e-mail address to which replies should be sent.

The `days` field can contain any of the following flags to specify which days of the week the mail daemon should automatically check mail for the described account:

Constant	Meaning
B_CHECK_NEVER	Don't automatically check the account's mail.
B_CHECK_WEEKDAYS	Check the mail only on weekdays.
B_CHECK_DAILY	Check the mail every day.
B_CHECK_CONTINUOUSLY	Check continuously every day.

The `interval` specifies how many seconds apart each e-mail retrieval should be, and the `begin_time` and `end_time` specify the time of day (in seconds) that automatic retrieval should begin and end. If `begin_time` and `end_time` are the same, the daemon checks mail round-the-clock.

Eventually these functions will support multiple POP accounts; at this time, the Mail Kit only supports one POP account, so you must use an *index* of 0. Any other index will result in a **B_BAD_INDEX** error.

`get_pop_account()` fills the specified **`mail_pop_account`** structure with the information on the POP account, and `set_pop_account()` takes the information in the buffer and saves it as the new default.

RETURN CODES

- **B_OK**. The notification was successfully set or retrieved.
- **B_BAD_INDEX**. An index other than 0 was specified.
- **B_NO_REPLY**. The mail daemon didn't reply to the request.

---

get_smtp_host() , set_smtp_host()

status_t **get_smtp_host**(char **smtp_host*)
status_t **set_smtp_host**(char **smtp_host*, bool *save* = true)

`get_smtp_host()` returns in the buffer pointed to by *smtp_host* the name of the SMTP host as currently configured. The buffer should be at lest **`B_MAX_HOST_NAME_LENGTH`** bytes long.

`set_smtp_host()` sets the SMTP host through which mail will be sent in the future to the specified host. If *save* is **`true`**, the new setting becomes the default and will persist through a reboot of the computer; otherwise, the change is only temporary.

RETURN CODES

- **`B_OK`**. The notification was successfully set or retrieved.
- **`B_NO_REPLY`**. The mail daemon didn't respond to the request.

---

send_queued_mail()

```
status_t send_queued_mail(void)
```

Tells the mail daemon to send all pending outgoing mail.

RETURN CODES

- **`B_OK`**. Mail transfer intitiated successfully.
- Errors from BMessage::**`SendMessage()`**.

The Be Book

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Mail Daemon Function Summary

Function Summary

Function	Description
check_for_mail()	Sends and retrieves mail
count_pop_accounts()	Returns the number of configured POP accounts
decode_base64()	Decodes base-64 encoded data
encode_base64()	Writes data in base-64 encoding
forward_mail()	Forwards a given mail message to a list of recipients
get_mail_notification()	Returns mail notification configuration information
get_pop_account()	Retrieves a given POP account's configuration information
get_smtp_host()	Retrieves the name of the SMTP host used by the mail daemon
send_queued_mail()	Sends all pending outgoing mail
set_mail_notification()	Sets mail notification configuration information
set_pop_account()	Sets a given POP account's configuration information
set_smtp_host()	Sets the name of the SMTP host used by the mail daemon

---

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Mail Kit

The Mail Kit provides Internet e-mail messaging services. These services include:

- Configuration of the user's mail accounts.
- Sending messages using the Simple Mail Transfer Protocol (SMTP).
- Receiving messages via the Post Office Protocol (POP).
- Automatic, timed sending and receiving of messages.
- Encoding and decoding base-64 encoded data.

An assortment of global C functions are provided by the Mail Kit to configure the mail daemon and process base-64 data. See "The Mail Daemon" for information on how to use these functions.

Outgoing mail messages are constructed and sent using the BMailMessage class.

Mail Message Files

Every mail message is stored in an individual file with attached attributes that describe the message in detail; you can query the file system to obtain information about the sender, subject, and receiver of the message, among other things. See "Querying Mail Messages" below for more detailed information and an example program.

Every message the user writes is saved in a file until it's sent by the mail daemon (and may or may not be deleted after being sent). Likewise, messages that the mail daemon has retrieved are also stored in files on a local disk.

The process of sending a mail message works something like this:

- The user writes a new mail message and chooses the mail writing program's "Send" option.
- The mail writing program creates a BMailMessage object and configures it based on the user's inputs by setting the "To," "Subject," and other header fields appropriately, and by storing the message content into the BMailMessage.
- The program then calls the BMailMessage object's **`Send()`** function to tell the mail daemon to send the message.
- The mail daemon creates a disk file that contains the message. The message content is stored in the file itself, and attributes are created to contain the "To," "Subject," and other relevant header fields. See "Querying Mail Messages" for more information. The message's status attribute is set to "New".
- The next time the mail daemon's **`check_for_mail()`** function is called (either automatically or explicitly), the daemon sends the message via SMTP, then changes the message's status attribute to "Sent". This is done for all mail messages whose status is "Pending".

After sending outgoing messages, the mail daemon will also check to see if any incoming mail is waiting to be retrieved. If there is, it proceeds something like this:

- The mail daemon fetches the first message from the mail server via POP.
- The daemon creates a new mail message file, and the message is written into the file. The file contains the Internet headers, message content, and all enclosures (if any).
- The mail daemon scans the message and adds attributes to the message file for each of the header fields, as well as a couple of extra attributes. These are described in detail in "Querying Mail Messages" below.
- The daemon continues reading mail messages from the mail server until there aren't any left.

Querying Mail Messages

The Mail Kit takes full advantage of the BeOS attribute and query system. Each message received is parsed by the mail daemon and important information about it is converted into a defined collection of attributes attached to the file. This makes it extremely easy to create applications that search for messages meeting specific parameters. In an example to follow shortly, we'll create a program that lists all unread messages.

Once the mail daemon has received a message and saved it to disk, any application can query the file system to locate messages that meet certain parameters, then read the attributes and message content to present information about that message to the user.

The following attributes are provided by the Mail Kit:

Constant	Attribute Name	Description
B_MAIL_ATTR_NAME	MAIL:name	Name of the mail file.
B_MAIL_ATTR_STATUS	MAIL:status	Message status.
B_MAIL_ATTR_PRIOIRITY	MAIL:priority	"Priority" field value.
B_MAIL_ATTR_TO	MAIL:to	"To" field value.
B_MAIL_ATTR_CC	MAIL:cc	"Cc" field value.
B_MAIL_ATTR_FROM	MAIL:from	"From" field value.
B_MAIL_ATTR_SUBJECT	MAIL:subject	"Subject" field value.
B_MAIL_ATTR_REPLY	MAIL:reply	"Reply-to" field value.
B_MAIL_ATTR_WHEN	MAIL:when	"When" field value.
B_MAIL_ATTR_FLAGS	MAIL:flags	Message flags.
B_MAIL_ATTR_RECIPIENTS	MAIL:recipients	List of message recipients.
B_MAIL_ATTR_MIME	MAIL:mime	The MIME version used.
B_MAIL_ATTR_HEADER	MAIL:header_length	Length of the message header.
B_MAIL_ATTR_CONTENT	MAIL:content_length	Length of the message content.

B_MAIL_ATTR_NAME is a string that identifies the name of the sender.

B_MAIL_ATTR_STATUS is a string that identifies the status of the message. Possible values are:

Status String	Description
"Error"	An error occurred trying to send the message.
"New"	The message has not been read yet.
"Pending"	The message has not been sent yet.
"Read"	The message has been read.
"Sent"	The message has been sent.

B_MAIL_ATTR_PRIORITY is a string that contains the value of the Priority field in the message.

B_MAIL_ATTR_TO, **B_MAIL_ATTR_FROM**, and **B_MAIL_ATTR_REPLY** are strings that contain the primary recipient's e-mail address, the sender's e-mail address, and the sender's reply-to address.

B_MAIL_ATTR_CC contains the addresses to whom the message is carbon copied.

B_MAIL_ATTR_SUBJECT is a string containing the subject of the message.

B_MAIL_ATTR_WHEN is a BeOS time field (**B_TIME_TYPE**) containing the message's "When" field.

B_MAIL_ATTR_FLAGS contains 32-bit integer (int32) flags which can be any combination of the following values:

Flag	Description
B_MAIL_PENDING	Message is waiting to be sent.
B_MAIL_SENT	Message has been sent.
B_MAIL_SAVE	Mail will be saved after being sent.

The **B_MAIL_ATTR_FLAGS** attribute is BeOS-specific.

B_MAIL_ATTR_RECIPIENTS contains a list of all recipients of the message (primary, cc, and bcc), but is only valid for outgoing messages (this attribute doesn't exist on incoming messages).

B_MAIL_ATTR_MIME contains a string that defines the version number of the MIME specification used to transmit any enclosures attached to the file. This attribute is only present if the file has one or more enclosures.

B_MAIL_ATTR_HEADER and **B_MAIL_ATTR_CONTENT** contain the lengths, in bytes, of the header and content portions of the message. They're both **int32** type data.

The following attributes are indexed:

- **B_MAIL_ATTR_NAME**
  B_MAIL_ATTR_STATUS
  B_MAIL_ATTR_PRIORITY
  B_MAIL_ATTR_TO
  B_MAIL_ATTR_CC
  B_MAIL_ATTR_FROM
  B_MAIL_ATTR_SUBJECT
  B_MAIL_ATTR_REPLY
  B_MAIL_ATTR_WHEN
  B_MAIL_ATTR_FLAGS

The headers, the contents of the messag,e and the enclosures (in base-64 encoded form) can all be found in the file itself and can be read using a BFile object. See "The Storage Kit" in the for further information on reading files.

Queries and Mail Messages

Now that you know what attributes are available on mail message files, and which attributes are indexed, you can consider all the clever things you can use them for. Let's look at an

example program that, from a Terminal window, lets you see a list of the unread mail you have.

```
void main(void) {
   BQuery query;
   BNode node;
   BVolume vol;
   BVolumeRoster vroster;
   entry_ref ref;
   char buf[256];
   int32 message_count = 0;

   vroster.GetBootVolume(&vol);
   query.SetVolume(&vol);
```

The program begins by establishing needed variables, then using a BVolumeRoster to set the query's search volume to the boot disk. This is covered in more detail in the Storage Kit chapter.

```
   if (query.SetPredicate("MAIL:status = New") != B_OK) {
      printf("Error: can't set query predicate.n");
      return;
   }
```

Then the query is configured to search for new mail. New messages can be identified by the **B_MAIL_ATTR_STATUS** attribute (called "MAIL:status") having a string value of "New". If an error occurs, the program prints an error end returns.

```
   if (query.Fetch() != B_OK) {
      printf("Error: new mail query failed.n");
      return;
   }
```

The query is told to fetch. Again, if this fails, an error message is displayed and the program returns.

```
   while (query.GetNextRef(&ref) == B_OK) {
      message_count++;      // Increment message counter
```

The loop scanning through the fetched new messages begins by incrementing the counter of new messages received.

```
      if (node.SetTo(&ref) != B_OK) {
         printf("Error: error scanning new messages.n");
         return;
      }
```

Then a BNode is set to reference the message file. If this fails, the program displays an error message and quits.

```
      buf[0] = '0';      // If error, use empty string
      node.ReadAttr(B_MAIL_ATTR_FROM, B_STRING_TYPE, 0, buf, 255);
      buf[20] = '0';      // Truncate to 20 characters
      printf("%3d From: %-20s", message_count, buf);
```

The buffer we're using the receive the attribute values is initialized to an empty string, then we call BNode's **ReadAttr()** function to read the **B_MAIL_ATTR_FROM** attribute into the buffer. We then truncate the read string to 20 characters for display purposes (to make it fit into the table we're outputting) and print the message number and sender information.

```
      buf[0] = '0';      // If error, use empty string
      node.ReadAttr(B_MAIL_ATTR_SUBJECT, B_STRING_TYPE, 0, buf, 255);
      buf[40] = '0';      // Truncate to 40 characters
      printf(" Sub: %sn", buf);
   }
```

The buffer is reset to an empty string and the **B_MAIL_ATTR_SUBJECT** attribute is read into it. This string is truncated to 40 characters, then printed.

This loop continues until no more new messages are found; this is detected when **GetNextRef()** returns an error.

```
   if (message_count) {
      printf("%d new messages.n", message_count);
   }
   else {
      printf("No new messages.n");
   }
}
```

Finally, the number of new messages is printed. If there aren't any messages, we very politely print "No new messages." rather than "0 new messages," for a little added panache.

This simple example demonstrates how you can use the attributes provided by the Mail Kit to create mail message reading applications. The message body and attachments are stored within the file itself, and can be read using the functions described in the BFile section in the Storage Kit chapter of the "Be Developer's Guide."

The E-mail File Type

The mail daemon ensures, when it's first launched at system boot time, that the BeOS File Type database includes an entry for e-mail files. E-mail files have the MIME string "text/x-email", which is represented by the constant **B_MAIL_TYPE**.

The E-mail entry in the File Type database includes a list of the attributes on which you can search. You can look up this information using the BMimeType class's **GetAttrInfo()** function:

```
BMimeType mime;
BMessage message;

mime.SetTo(B_MAIL_TYPE);
mime.GetAttrInfo(&message);
```

After running this code, the *message* contains the description of the attributes available on e-mail files. The message has three useful arrays of items:

Message Item	Description
"attr:public_name"	The user-readable name of the attribute.
"attr:name"	The attribute name used for BNode and `fs_attr_*` calls.
"attr:type"	The type of data the attribute contains.

You can examine the items in each of these arrays in the message to get useful information about the attributes you can reference on the e-mail files. For example:

```
printf("Public name: %sn", FindString("attr:public_name", 1));
printf("Name: %sn", FindString("attr:name", 1));
```

This code will print the public name and the attribute name of the second attribute registered in the File Type database entry for e-mail files. In the current implementation of the Mail Kit, this would print:

```
Public name: Subject
Name: MAIL:subject
```

The number of (and order of) attributes in the database entry might change in the future—and that's where the File Type database comes in handy. If, a year from now, Be adds more attributes to e-mail files, your File Type database-savvy application won't have to be updated to support them.

At this time, the File Type database has attribute information records for each of the following attributes:

- **B_MAIL_ATTR_NAME**
  B_MAIL_ATTR_STATUS

B_MAIL_ATTR_PRIORITY
B_MAIL_ATTR_TO
B_MAIL_ATTR_FROM
B_MAIL_ATTR_SUBJECT
B_MAIL_ATTR_REPLY
B_MAIL_ATTR_WHEN

You can obtain information on these attributes, their formats, and their user-readable names by looping through the arrays in the message until the BMessage::`Find...()` function returns **NULL** or **B_BAD_INDEX**.

For more information on the BMimeType class and the File Type database, see BMimeType in The Storage Kit.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Network Kit Table of Contents

- BNetAddress
- BNetBuffer
- BNetDebug
- BNetEndpoint
- Network Sockets

The Be Book **The Network Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The Network Kit: Master Index

=

=	BNetBuffer
=	BNetEndpoint

A

Accept()	BNetEndpoint
accept()	Network Sockets
AppendData()	BNetBuffer
AppendDouble()	BNetBuffer
AppendFloat()	BNetBuffer
AppendInt16()	BNetBuffer
AppendInt32()	BNetBuffer
AppendInt64()	BNetBuffer
AppendInt8()	BNetBuffer
AppendMessage()	BNetBuffer
AppendString()	BNetBuffer
AppendUint16()	BNetBuffer
AppendUint32()	BNetBuffer
AppendUint64()	BNetBuffer
AppendUint8()	BNetBuffer
Archiving BNetEndpoints	BNetEndpoint

The Arguments	Network Sockets

B

The bind() Arguments	Network Sockets
Bind()	BNetEndpoint
bind()	Network Sockets
BytesRemaining()	BNetBuffer

C

Close()	BNetEndpoint
closesocket()	Network Sockets
Connect()	BNetEndpoint
connect()	Network Sockets
Constructor and Destructor	BNetAddress
Constructor and Destructor	BNetBuffer
Constructor and Destructor	BNetEndpoint

D

Data()	BNetBuffer
Dump()	BNetDebug

E

Enable()	BNetDebug

Error()	BNetEndpoint
ErrorStr()	BNetEndpoint

F

Functions	Network Sockets

G

GetAddr()	BNetAddress
getpeername()	Network Sockets
getsockname()	Network Sockets

I

InitCheck()	BNetAddress
InitCheck()	BNetEndpoint
IsDataPending()	BNetEndpoint
IsEnabled()	BNetDebug

L

Listen()	BNetEndpoint
listen()	Network Sockets
LocalAddr()	BNetEndpoint

M

Member Functions	BNetAddress
Member Functions	BNetBuffer
Member Functions	BNetEndpoint

N

BNetAddress	BNetAddress
BNetAddress()	BNetAddress
~BNetAddress	BNetAddress
BNetBuffer	BNetBuffer
BNetBuffer()	BNetBuffer
~BNetBuffer	BNetBuffer
BNetDebug	BNetDebug
BNetDebug	BNetDebug
BNetEndpoint	BNetEndpoint
BNetEndpoint()	BNetEndpoint
~BNetEndpoint	BNetEndpoint
Network Sockets	Network Sockets
Network Sockets	Network Sockets

O

Operators	BNetBuffer
Operators	BNetEndpoint

P

Print()	BNetDebug

R

Receive()	BNetEndpoint
ReceiveFrom()	BNetEndpoint
recv()	Network Sockets
recvfrom()	Network Sockets
RemoteAddr()	BNetEndpoint
RemoveData()	BNetBuffer
RemoveDouble()	BNetBuffer
RemoveFloat()	BNetBuffer
RemoveInt16()	BNetBuffer
RemoveInt32()	BNetBuffer
RemoveInt64()	BNetBuffer
RemoveInt8()	BNetBuffer
RemoveMessage()	BNetBuffer
RemoveString()	BNetBuffer
RemoveUint16()	BNetBuffer
RemoveUint32()	BNetBuffer
RemoveUint64()	BNetBuffer
RemoveUint8()	BNetBuffer

S

select()	Network Sockets

Send()	BNetEndpoint
send()	Network Sockets
SendTo()	BNetEndpoint
sendto()	Network Sockets
SetNonBlocking()	BNetEndpoint
SetOption()	BNetEndpoint
SetProtocol()	BNetEndpoint
SetReuseAddr()	BNetEndpoint
SetTimeout()	BNetEndpoint
SetTo()	BNetAddress
setsockopt()	Network Sockets
Size()	BNetBuffer
The socket() Arguments	Network Sockets
Socket()	BNetEndpoint
socket()	Network Sockets
Sorts of Sockets	Network Sockets
Static Functions	BNetDebug

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetBuffer

Derived from: BArchivable
Declared in: be/net/NutBuffer.h
Library: libnetapi.so
Allocation: Constructor only
Summary: *more...*

The BNetBuffer class provides an easy way to construct network buffers consisting of any sort of data, for use by the BNetEndpoint class.

Once you've created a BNetBuffer, you can append data to it by using a series of functions designed to add various types of data. For example, to create a buffer and place the long integer 2 followed by the string "This is a test." in it, you could do this:

```
BNetBuffer buffer(512);
buffer.AppendInt32(2);
buffer.AppendString("This is a test.");
```

The **AppendInt32()** function automatically handles conversion of the value into network byte order, as do all of the **AppendXXX()** functions for integer values (16-bit, 32-bit, and 64-bit, signed or unsigned). Likewise, the **RemoveXXX()** functions peel data out of a buffer, and they too are endian-aware.

Constructor and Destructor

BNetBuffer()

```
BNetBuffer(size_t size = 0)
BNetBuffer(const BNetBuffer &from)
BNetBuffer(BMessage *archive)
```

Creates a BNetBuffer. The first form creates a new buffer capable of holding up to size bytes of data. In this case, the buffer begins life empty.

The second form creates a new buffer which is an exact duplicate of the BNetBuffer specified by the *from* argument, including any data that might already be in the buffer. The third form reconstructs an archived BNetBuffer.

~BNetBuffer

```
virtual ~BNetEndpoint()
```

A typical destructor.

Member Functions

AppendInt8() , AppendUint8() , AppendInt16() , AppendUint16() , AppendInt32() , AppendUint32() , AppendInt64() , AppendUint64() , AppendFloat() , AppendDouble() , AppendString() , AppendData() , AppendMessage()

status_t **AppendInt8(**int8 *value***)**
status_t **AppendUint8(**uint8 *value***)**
status_t **AppendInt16(**int16 *value***)**
status_t **AppendUint16(**uint16 *value***)**
status_t **AppendInt32(**int32 *value***)**
status_t **AppendUint32(**uint32 *value***)**
status_t **AppendInt64(**int64 *value***)**
status_t **AppendUint64(**uint64 *value***)**
status_t **AppendFloat(**float *value***)**
status_t **AppendDouble(**double *value***)**
status_t **AppendString(**const char **string***)**
status_t **AppendData(**const void **data*, size_t *size***)**
status_t **AppendMessage(**BMessage &*message***)**

Appends the specified data type to the end of the buffer. Integer values are automatically converted to network byte ordering (but floats and doubles are not, nor are values inside structures added using `AppendData()`).

Strings are appended as null-terminated strings.

`AppendData()` copies *size* bytes from the buffer pointed at by *data*.

RETURN CODES

- B_OK. The data was appended without error.
- **B_ERROR**. The data couldn't be appended.

Data() , Size() , BytesRemaining()

unsigned char ***Data(**void**)** const
size_t **Size(**void**)** const
size_t **BytesRemaining(**void**)** const

`Data()` returns a pointer to the BNetBuffer's internal data buffer.

`Size()` returns the number of bytes currently in the buffer.

`BytesRemaining()` returns the number of unused bytes in the buffer.

RemoveInt8() , RemoveUint8() , RemoveInt16() , RemoveUint16() , RemoveInt32() , RemoveUint32() , RemoveInt64() , RemoveUint64() , RemoveFloat() , RemoveDouble() , RemoveString() , RemoveData() , RemoveMessage()

status_t **RemoveInt8(**int8 &*value*)
status_t **RemoveUint8(**uint8 &*value*)
status_t **RemoveInt16(**int16 &*value*)
status_t **RemoveUint16(**uint16 &*value*)
status_t **RemoveInt32(**int32 &*value*)
status_t **RemoveUint32(**uint32 &*value*)
status_t **RemoveInt64(**int64 &*value*)
status_t **RemoveUint64(**uint64 &*value*)
status_t **RemoveFloat(**float &*value*)
status_t **RemoveDouble(**double &*value*)
status_t **RemoveString(**char **string*, size_t *size*)
status_t **RemoveData(**void **data*, size_t *size*)
status_t **RemoveMessage(**BMessage &*message*)

Removes the specified data type from the buffer. Integer values are automatically converted from network byte ordering (but floats and doubles are not, nor are values inside structures removed using `RemoveData()`).

Strings are removed as null-terminated strings, up to *size* bytes. Be sure the string buffer is at least *size* bytes long.

`RemoveData()` removes *size* bytes and copies them into the buffer pointed at by *data*. Be sure the *data* buffer is at least *size* bytes long.

These functions start at the beginning of the buffer. After each item is removed, the next `Remove...()` call will start at the next byte in the buffer.

RETURN CODES

- B_OK. The data was removed without error.
- **B_ERROR**. The data couldn't be removed.

Size() see Data()

Operators

= (assignment)

```
BNetBuffer &operator =(const BNetBuffer &from)
```

Copies the BNetBuffer specified by *from* into the left-side object, thereby duplicating that object. If *from* is connected, the left-side object will duplicate and open the same connection. Even the data in the buffer is copied, if there is any.

The Be Book The Network Kit The Network Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetBuffer Function Summary

Function Summary

Function	Description
=	Makes one BNetBuffer identical to another
AppendInt8()	Appends data of the specified type to the end of the buffer
BytesRemaining()	Returns the number of unused bytes left in the buffer
Data()	Returns a pointer to the BNetBuffer's internal data buffer
BNetBuffer()	Constructor
RemoveInt8()	Removes data of the specified type from the beginning of the buffer
Size()	Returns the number of bytes currently in the buffer

The Be Book **The Network Kit** **The Network Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetEndpoint

Derived from: BArchivable
Declared in: be/net/NetEndpoint.h
Library: libnetapi.so
Allocation: Constructor only
Summary: *more...*

The BNetEndpoint class represents a network endpoint, which can send and receive data, establish network connections, bind to local addresses, and listen for and accept new connections.

Rather than replacing the existing network architecture, the BNetEndpoint class provides a C++ wrapper to the standard socket functions. All the same rules of usage apply, so be sure to review the BSD-like C socket function material.

Archiving BNetEndpoints

BNetEndpoint objects are archivable. All attributes of the BNetEndpoint are preserved when archived. Upon reinstantiation, the object is duplicated precisely. This has interesting ramifications—if the BNetEndpoint is connected to a remote system when it's archived, the reinstantiated archive will also be connected to that system. You can actually archive active connections to restore them later.

Obviously, however, protocol-specific information won't be saved unless you add that data to the archive yourself. For example, if you archive an FTP connection, then restore the connection from the archive, the working directory or any ongoing downloads won't be restored automatically.

Constructor and Destructor

BNetEndpoint()

```
BNetEndpoint(int protocol = SOCK_STREAM)
BNetEndpoint(const BNetEndpoint &)
BNetEndpoint(BMessage *archive)
```

Creates a BNetEndpoint representing a network connection endpoint on the local system. After construction, you must call **InitCheck()** to ensure that no errors occurred during setup.

The *protocol* argument lets you specify whether the BNetEndpoint will use a stream socket (**SOCK_STREAM**) or a datagram socket (**SOCK_DGRAM**).

By default, I/O is blocking and address reusing is off. You can change these by calling **SetNonBlocking()** and **SetReuseAddr()**.

~BNetEndpoint

```
virtual ~BNetEndpoint()
```

A typical destructor.

Member Functions

Accept() see Listen()

Bind()

virtual status_t **Bind(**const BNetAddress &*address***)**
virtual status_t **Bind(**int *port* = 0**)**

Binds the BNetEndpoint to a specific local address. That address can be specified by using a BNetAddress or a simple *port* number. This selects the port that will handle the local end of the connection.

If your BNetEndpoint is using a stream protocol and is going to be listening for connections, you must call `Bind()`.

If your stream BNetEndpoint is a client, it doesn't have to call `Bind()` but you can if you want to. There aren't any significant benefits to doing so, however.

A stream accept BNetEndpoints must not be bound.

Datagram BNetEndpoints that are going to receive data must be bound; datagram BNetEndpoints that will only be sending data don't have to be. However, if an endpoint will both send and receive, it must be bound.

If you don't specify an address or port number, or specify a port number of 0, `Bind()` will bind the BNetEndpoint to a random local port. You can determine which one by calling **`LocalAddr()`**.

The only way to unbind a BNetEndpoint from an address or port is to close the endpoint.

RETURN CODES

- B_OK. The address was successfully bound to.
- **`B_ERROR`**. An error occurred.

Close()

virtual void **Close(**void**)**

Closes the connection, if there is one. If there's unread data buffered up, it's disposed of.

Connect()

virtual status_t **Connect(**const BNetAddress &*address***)**

virtual status_t **Connect(**const char *_address_, int _port_**)**

Opens a connection to the specified remote system. The system's address can be specified by either using a BNetAddress or by specifying the IP address or domain name and port number. For example, to connect to the Megaburger, Inc. web server, your software would call:

```
status_t err = myEndpoint->Connect("www.megaburger.com", 80);
if (err != B_OK) {
   /* error occurred */
}
else {
   /* all is well, connection is open */
}
```

RETURN CODES

- B_OK. The connection was opened.
- **B_ERROR**. An error occurred.

Error() , ErrorStr()

int Error(void) const

char ***ErrorStr(**void**)** const

`Error()` returns the integer error code for the last send or receive error. If you receive a **B_ERROR** result from a send or receive function, you can find out the specific error using this function.

`ErrorStr()` returns a pointer to a text string describing the error; this string isn't yours, so **don't** try to **free()** it.

InitCheck()

status_t **InitCheck(**void**)** const

Returns a status_t indicating whether or not the object was successfully instantiated.

RETURN CODES

- B_OK. The BNetEndpoint was constructed without error.
- **B_ERROR**. An error occurred during construction.

IsDataPending()

virtual bool **IsDataPending(**bigtime_t *timeout* = 0**)**

Returns **true** if there's data waiting to be received, otherwise returns **false**. If you specify a *timeout* other than 0, the function will block until either data is available or the timeout period elapses.

Listen() , Accept()

virtual status_t **Listen(**int *backlog* = 5**)**

virtual BNetEndpoint ***Accept(**int32 *timeout* = -1**)**

`Listen()` tells the BNetEndpoint to begin listening for incoming connection attempts on its local port. These attempts are queued; up to *backlog* attempts can be in the queue at any time. If more attempts are backlogged than that, the later attempts will be rejected until there's room in the queue.

You can accept an incoming connection attempt by calling `Accept()`. If there are no connection attempts queued up, this function returns **NULL**. If there are connection attempts in the queue, a new BNetEndpoint object is created with the connection between your local port and the remote system opened, and that BNetEndpoint is returned to you.

The new connection is yours to do with as you please. When you're finished with the connection, you must delete the returned BNetEndpoint. Typically your listener thread will look something like this:

```
long MyListener(void *data) {
   BNetEndpoint endpoint;

   if (endpoint.InitCheck() < B_OK) {
      return -1;
   }

   endpoint.Bind(portNumber);   /* bind to the desired port */
   endpoint.Listen();   /* listen for connections */

   while (keepListening) {
      BNetEndpoint *connect = NULL;
      connect = endpoint.Accept();
      if (connect) {
         handle_connection(connect, data);   /* call a function do the work */
         delete connect;
      }
   }
   endpoint.Close();
}
```

RETURN CODES

- B_OK. Success.
- **B_ERROR**. Failure.

LocalAddr() , RemoteAddr()

const BNetAddress &**LocalAddr**(void)
const BNetAddress &**RemoteAddr**(void)

These functions return a BNetAddress representing the local or remote system on the connection. `LocalAddr()` returns the address of the local machine, and `RemoteAddr()` (amazingly enough) returns the address of the remote system.

If there isn't a remote connection, `RemoteAddr()` will return a BNetAddress indicating an IP address of 0.0.0.0.

Receive() , ReceiveFrom()

virtual int32 **Receive**(const void **buffer*, size_t *size*, int *flags* = 0) virtual int32 **Receive**(BNetBuffer &*buffer*, size_t *size*, int *flags* = 0)
virtual int32 **ReceiveFrom**(const void **buffer*, size_t *size*, const BNetAddress &*from*, int *flags* = 0) virtual int32 **ReceiveFrom**(BNetBuffer &*buffer*, size_t *size*, const BNetAddress &*from*, int *flags* = 0)

`Receive()` receives a *buffer* of data from the remote end of the connection. If there's no connection established, **B_ERROR** is returned immediately. Up to *size* bytes of data are received.

ReceiveFrom() receives the buffer from the remote system specified by the *from* BNetAddress. **ReceiveFrom()** only works if the connection is using a datagram protocol.

The first form of each function function sends an arbitrary *buffer* of the specified *size*, and the second form sends the contents of a BNetBuffer. When using a BNetBuffer, incoming data is appended to the end of the buffer, so you can use the same buffer in a loop to buffer incoming data in chunks until the desired number of bytes have been read.

The *flags* argument, which is passed on to the **socket.h** **recv()** or **recvfrom()** function, is currently unused and must be 0.

When you call these functions in blocking mode (which is the default), they block until there's data available to receive or a timeout occurs. The timeout period is set by calling **SetTimeout()**. You can turn on or off blocking by calling **SetNonBlocking()**. If you're in nonblocking mode and there's no data waiting, these functions return 0 immediately, indicating that there's no data.

These functions return the number of bytes actually received, or -1 if an error occurred. You can call **Error()** to get the specific error that occurred.

RemoteAddr() see LocalAddr()

Send() , SendTo()

virtual int32 **Send(**const void **buffer*, size_t *size*, int *flags* = 0**)** virtual int32 **Send(**BNetBuffer &*buffer*, int *flags* = 0**)**
virtual int32 **SendTo(**const void **buffer*, size_t *size*, const BNetAddress &*to*, int *flags* = 0**)** virtual int32 **SendTo(**BNetBuffer &*buffer*, const BNetAddress &*to*, int *flags* = 0**)**

`Send()` sends a *buffer* of data to the remote end of the connection. If there's no connection established, **B_ERROR** is returned immediately. In addition, if the BNetEndpoint is configured to use a datagram protocol, this function fails unless **Connect()** has been called, since that function caches the destination address.

SendTo() sends the buffer to the remote system specified by the *to* BNetAddress. **SendTo()** only works if the connection is using a datagram protocol.

The first form of each function function sends an arbitrary *buffer* of the specified *size*, and the second form sends the contents of a BNetBuffer.

The *flags* argument, which is passed on to the **socket.h** **send()** or **sendto()** function, is currently unused and must be 0.

These functions return the number of bytes actually sent, or -1 if an error occurred. You can call **Error()** to get the specific error that occurred.

SetOption() , SetNonBlocking() , SetReuseAddr()

int **SetOption(**int32 *option*, int32 *level*, const void **data*, unsigned int *dataSize***)**
int **SetNonBlocking(**bool *enable* = true**)**
int **SetReuseAddr(**bool *enable* = true**)**

`SetOption()` lets you set any option for the BNetEndpoint. This provides access to the **setsockopt()** function of the underlying socket.

`SetNonBlocking()` controls whether I/O should block or not. If *enable* is **true**, the connection will be nonblocking. If *enable* is **false**, the connection will block on I/O calls until the transmission is completed.

`SetReuseAddr()` controls whether addresses should be reused or not. If *enable* is **true**, addresses will be reused. If *enable* is **false**, the connection won't reuse addresses.

These functions return 0 if successful; otherwise they return -1.

SetProtocol()

```
status_t SetProtocol(int protocol)
```

Sets the protocol type for the connection. Possible values for the *protocol* argument are **SOCK_STREAM** (to use the stream protocol) or **SOCK_DGRAM** (for datagram protocol).

RETURN CODES

- B_OK. The protocol was set successfully.
- **B_ERROR**. An error occurred setting the protocol.

SetTimeout()

```
void SetTimeout(bigtime_t timeout)
```

Sets the timeout for calls to **Receive()** and **ReceiveFrom()**. If blocking I/O is in use, and *timeout* microseconds pass, the function will abort with an error. By default, there's no timeout. You can specify that you want no timeout by specifying -1.

Socket()

```
int Socket(void) const
```

Returns the actual socket used by the BNetEndpoint for data communications.

Operators

= (assignment)

BNetEndpoint &**operator =**(const BNetEndpoint &*from*)

Copies the BNetEndpoint specified by *from* into the left-side object, thereby duplicating that object. If *from* is connected, the left-side object will duplicate and open the same connection.

The Be Book The Network Kit The Network Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetEndpoint Function Summary

Function Summary

Function	Description
=	Makes one BNetEndpoint identical to another
Accept()	Selects blocking or nonblocking I/O mode
Bind()	Binds the BNetEndpoint to a specific local address
Close()	Closes the connection, if one exists, and discards unread data
Connect()	Establishes a connection to a remote computer
Error()	Returns the last send/receive error code
ErrorStr()	Returns a string describing the last error
InitCheck()	Indicates whether or not the object was initialized successfully
IsDataPending()	Returns true if there's data waiting to be received
Listen()	Sets endpoint options
LocalAddr()	Sets endpoint options
BNetEndpoint()	Constructor
Receive()	Receives data from the remote computer
ReceiveFrom()	Receives data from the specified network address
RemoteAddr()	Selects blocking or nonblocking I/O mode
Send()	Sends data to the remote end of the connection

SendTo()	Sends data to the specified destination address
SetNonBlocking()	Selects blocking or nonblocking I/O mode
SetOption()	Sets endpoint options
SetProtocol()	Sets the protocol type for the connection
SetReuseAddr()	Selects or disables address reusing mode
SetTimeout()	Sets the optional timeout for calls to Receive() and ReceiveFrom()
Socket()	Returns the socket used for connections on this BNetEndpoint

---

The Be Book **The Network Kit** **The Network Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetAddress

Derived from: BArchivable
Declared in: be/net/NetAddress.h
Library: libnetapi.so
Allocation: Constructor only
Summary: *more...*

The BNetAddress class is used to construct and represent network addresses, providing access to them in a variety of formats. Additional functions are available to convert addresses from one format into another.

Constructor and Destructor

BNetAddress()

BNetAddress(const char *_hostname_ = NULL, unsigned short _port_ = 0**)**
BNetAddress(const sockaddr_in &_sa_**)**
BNetAddress(in_addr _addr_, int _port_ = 0**)**
BNetAddress(uint32 _addr_, int _port_ = 0**)**
BNetAddress(const char *_hostname_, const char *_protocol_, const char *_service_**)**
BNetAddress(BNetAddress &**)**
BNetAddress(BMessage *_archive_**)**

Sets up the BNetAddress object to refer to the specified address. The address can be specified in a number of ways:

- By *hostname* and *port* number. For example, to connect to the HTTP port at www.be.com, you would specify "www.be.com" as the *hostname*, and 80 for the *port* number.
- By sockaddr_in structure. This structure contains the network family, port number, and IP address that make up the address.
- By IP address and port number. The IP address can be specified either using an in_addr, or by using a uint32. The IP address must be specified in network byte order. See the **htonl()** function.
- By *hostname*, *protocol*, and *service* name. This causes the port to be looked up in the **/etc/services** file by matching the protocol (typically "tcp" or "udp") and service name (such as "http" or "ftp") against the entries in the file. See **getservbyname()** for details on the format of this file.
- By copying an existing BNetAddress.
- By unflattening an archived BNetAddress from a BMessage.

After creating your BNetAddress, you must call **InitCheck()** to ensure that no errors occurred during setup. You can change the address later by calling **SetTo()**.

~BNetAddress

virtual ~**BNetAddress()**

A typical destructor.

Member Functions

InitCheck()

```
status_t InitCheck(void) const
```

Returns a status_t indicating whether or not the object was successfully instantiated.

RETURN CODES

- B_OK. The BNetAddress was constructed without error.
- **B_ERROR**. An error occurred during construction.

GetAddr()

```
status_t GetAddr(char *hostname = NULL, unsigned short *port = NULL) const
status_t GetAddr(struct sockaddr_in &sa) const
status_t GetAddr(in_addr &addr, unsigned short *port = NULL) const
```

Returns the address represented by the BNetAddress object in the format indicated by the form of the function used.

If you don't care about the hostname (in the first form), you can specify **NULL** for the *hostname* argument; if you don't care about the port number, you can specify **NULL** for the *port* argument.

RETURN CODES

- B_OK. The address was returned successfully.
- **B_ERROR**. An error occurred fetching the address information.

SetTo()

```
status_t SetTo(const char *hostname, const char *protocol,
    const char *service)
status_t SetTo(const char *hostname = NULL, unsigned short port = 0)
status_t SetTo(const sockaddr_in &sa)
status_t SetTo(in_addr addr, int port = 0)
status_t SetTo(uint32 addr = INADDR_ANY, int port = 0)
```

Sets the address represented by the BNetAddress object in the format indicated by the form of the function used. These formats are described in the discussion of the constructor.

RETURN CODES

- B_OK. The address was set successfully.
- **B_ERROR**. An error occurred setting the address information.
-

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetAddress Function Summary

Function Summary

Function	Description
GetAddr()	Returns the address in the desired format
InitCheck()	Indicates whether or not the object was initialized successfully
BNetAddress()	Constructor
SetTo()	Sets the address represented by the object

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Network Sockets

Declared in: be/kit/net/socket.h
Library: libnet.so
more...

A socket is an entry onto a network. To transmit data to another computer, you create a socket, tell it how to find the other computer, and then tell it to send. To receive data, you create a socket, tell it which computer to listen to (in some cases), and then wait for data to come pouring in.

The socket story starts with the **socket()** function. The set of functions you need to call after that depends on the type of socket you're creating (as explained in **socket()**).

Functions

socket() , closesocket()

int **socket(**int *family*, int *type*, int *protocol***)**
int **closesocket(**int *socket***)**

The `socket()` function returns a token (a non-negative integer) that represents the local end of a connection to another machine (0 is a valid socket token).

Socket tokens are *not* file descriptors (this violates the BSD tradition).

Freshly returned, a socket token is abstract and unusable; to put the token to use, you have to pass it as an argument to other functions—such as `bind()` and `connect()`—that know how to establish a connection over the network. The function's arguments, which are examined in detail in "The socket() Arguments", accept these values:

Argument	Acceptable Values
family	**AF_INET**
type	**SOCK_STREAM**, **SOCK_DGRAM**
protocol	**0**, **IPPROTO_TCP**, **IPPROTO_UDP**, **IPPROTO_ICMP**

The most typical socket calls are:

```
/* Create a stream TCP socket. */
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0);

/* Create a datagram UDP socket. */
int udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

ICMP messages are normally sent through "raw" sockets; however, the Network Kit doesn't currently support raw sockets, so you should use a datagram socket instead:

```
/* Create a datagram icmp socket. */
long icmp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

`closesocket()` closes a socket's connection (if it's the type of socket that can hold a connection) and frees the resources that have been assigned to the socket. When you're done with the sockets that you've created, you should pass each socket token to `closesocket()`. No socket is exempt from the need to be closed. This extends to sockets that are created for you by the **`accept()`** function.

The socket() Arguments

`socket()`'s three arguments, all of which take predefined constants as values, describe the type of communication the socket can handle:

- *family* describes the network address format that the socket understands. Currently, it must be **`AF_INET`** (the Internet address format).
- *type* describes the persistence of the connection that can be formed through this socket. It must be either **`SOCK_STREAM`** or **`SOCK_DGRAM`**. **`SOCK_STREAM`** means the connection (which is formed through a **`connect()`** or **`bind()`** call) remains open until told to close. **`SOCK_DGRAM`** describes a *datagram* socket that's only open while data is being sent or received (typically through **`sendto()`** and **`recvfrom()`**). It's closed at all other times. Keep in mind that you still have to call `closesocket()` on a datagram socket when you're done with it.
- *protocol* describes the messaging protocol, which is closely related to the socket type. Although there are four acceptable values (0, **`IPPROTO_TCP`**, **`IPPROTO_UDP`**, and **`IPPROTO_ICMP`**), the only values that you should actually use are 0 or **`IPPROTO_ICMP`**. 0 tells the socket to choose the correct protocol based on the socket type: If you set the *type* to **`SOCK_STREAM`**, then a *protocol* of 0 automatically sets the messaging protocol to **`IPPROTO_TCP`**. Similarly, **`IPPROTO_UDP`** is the correct protocol for **`SOCK_DGRAM`**. It's an error to ask for a "udp stream" or a "tcp datagram."

Sorts of Sockets

There are only two socket type constants: **`SOCK_STREAM`** and **`SOCK_DGRAM`**. However, if we look at the way sockets are used, we see that there are really five different categories of sockets, as illustrated below.

The labelled ovals represent individual computers that are attached to the network. The solid circles represent individual sockets. The numbers near the sockets are keys to the socket categories, which are:

1. **The stream listener socket.** A stream listener socket provides access to a service that's running on the "listener" machine (you might want to think of the machine as a "server.") The listener socket waits for client machines to "call in" and ask to be served. In order to listen for clients, the listener must call **`bind()`**, which "binds" the socket to an IP address and machine-specific port, and then **`listen()`**. Thus primed, the socket waits for a client message to show up by sitting in an **`accept()`** call.

2. **The stream client socket**. A stream client socket asks for service from a server machine by attempting to connect to the server's listener socket. It does this through the **`connect()`** function. A stream client can be bound (you can call **`bind()`** on it), but it's not mandatory.

3. **The "accept" socket**. When a stream listener hears a client in an **`accept()`** call, the function call creates yet another socket called the "accept" socket. Accept sockets are valid sockets, just like those you create through **`socket()`**. In particular, you have to remember to close accept sockets (through **`closesocket()`**) just as you would the sockets you explicitly create. Note that you can't bind an accept socket—the socket is bound automatically by the system.

4. **The datagram receiver socket**. A datagram receiver socket is sort of like a stream listener: It calls **bind()** and waits for "senders" to send messages to it. Unlike the stream listener, the datagram receiver doesn't call **listen()** or **accept()**. Furthermore, when a datagram sender sends a message to the receiver, there's no ancillary socket created to handle the message (there's no UDP analog to the TCP accept socket).

5. **The datagram sender socket**. A datagram sender is the simplest type of socket—all it has to do is identify a datagram receiver and send messages to it, through the **sendto()** function. Binding a datagram sender socket is optional.

TCP communication is two-way. Once the link between a client and the listener has been established (through **bind()**/**listen()**/**accept()** on the listener side, and **connect()** on the client side), the two machines can talk to each other through respective and complementary **send()** and **recv()** calls.

Communication along a UDP path, on the other hand, is one-way. The datagram sender can send messages (through **sendto()**), and the datagram receiver can receive them (through **recvfrom()**), but the receiver can't send message back to the sender. However, you can simulate a two-way UDP conversation by binding both sockets. This doesn't change the definition of the UDP path, or the capabilities of the two types of datagram sockets, it simply means that a bound datagram socket can act as a receiver (it can call **recvfrom()**) or as a sender (it can call **sendto()**).

To be complete, it should be mentioned that datagram sockets can also invoke **connect()** and then pass messages through **send()** and **recv()**. The datagram use of these functions is a convenience; its advantages are explained in the description of the **sendto()** function.

RETURN CODES

Upon failure, `socket()` returns a negative value and sets **errno** to...

- **EAFNOSUPPORT.**,*format* was other than **AF_INET.**
- **EPROTOTYPE.**,*type* and *protocol* mismatch.
- **EPROTONOSUPPORT.**,Unrecognized *type* or *protocol* value.

`closesocket()` returns a negative value if its argument is invalid.

---

bind()

int **bind(**int *socket*, const struct sockaddr **interface*, int *size***)**

The `bind()` function creates an association between a socket and an "interface," where an interface is a combination of an IP address and a port number. Binding is, primarily, useful for receiving messgaes: When a message sender (whether it's a stream client or a datagram sender) sends a message, it tags the message with an IP address and a port number. The

receiving machine—the machine with the tagged IP address—delivers the message to the socket that's bound to the tagged port.

The necessity of the bind operation depends on the type of socket; referring to the five categories of sockets enumerated in the `socket()` function description (and illustrated in the charming diagram found there), the "do I need to bind?" question is answered thus:

1. **Stream listener sockets must be bound**. Furthermore, after binding a listener socket, you must then call `listen()` and, when a client calls, `accept()`.

2. **Stream client sockets can be bound**, but they don't have to be. If you're going to bind a client socket, you should do so *before* you call `connect()`. The advantages of binding a stream client escape me at the moment. In any case, the client doesn't have to bind to the same port number as the listener—the listener's binding and the client's binding are utterly separate entities (let alone that they are on different machines). However, the client does *connect* to the interface that the listener is bound to.

3. **Stream attach sockets must not be bound**.

4. **Datagram receiver sockets must be bound.**

5. **Datagram sender sockets don't *have to be bound***...but if you're going to turn around and use the socket as a receiver, then you'll have to bind it.

Once you've bound a socket, you can't unbind it. If you no longer want the socket to be bound to its interface, the only thing you can do is close the socket (`closesocket()`) and start all over again.

Also, a particular interface can be bound by only one socket at a time and a single socket can only bind to one interface at a time. If your socket needs to bind to more than one interface, you need to create more than one socket and bind each one separately. An example of this is given later in this function description.

The 1-to-1 binding differs with the BSD socket implementation, which expects a socket to be able to bind to more than one interface. Consider it a bug that will be fixed in a subsequent release.

The bind() Arguments

`bind()`'s first argument is the socket that you're attempting to bind. This is, typically, a socket of type **`SOCK_STREAM`**. The *interface* argument is the address/port combination (or "interface") to which you're binding the socket. The argument is typed as a **`sockaddr`** structure, but, in reality, you have to create and pass a **`sockaddr_in`** structure cast as a **`sockaddr`**. The **`sockaddr_in`** structure is defined as:

```
struct sockaddr_in {
   unsigned short sin_family;
   unsigned short sin_port;
   struct in_addr sin_addr;
```

```
    char sin_zero[4];
};
```

- `sin_family` is the same as the address format constant that used to create the socket (the first argument to `socket()`). **Currently, it's always AF_INET.**

- `sin_port` is the port number that the socket will bind to, given in network byte order. Valid port numbers are between 1 and 65535; numbers up to 1024 are reserved for services such as **ftp** and **telnet**. If you're not implementing a standard service, you should choose a port number greater than 1024. The actual value of the port number is meaningless, but keep in mind that the port number must be unique for a particular address; only one socket can be bound to a particular address/port combination.

Currently, there's no system-defined mechanism for allowing a client/sender machine to ask a listener/receiver machine for its port numbers. Therefore, when you create a networked application, you either have to hard-code the port numbers or, better yet, provide default port numbers that the user (or a system administrator) can easily change.

- `sin_addr` is an **`in_addr`** structure that stores, in its `s_addr` field, the IP address of the socket's machine. As always, the address is in network byte order. You can use an address of 0 to tell the binding mechanism to find an address for you. By convention, binding to address 0 (which is conveniently symbolized by the **`INADDR_ANY`** address) means that you want to bind to *every* address by which your computer is known, including the "loopback" (address 127.0.0.1, or the constant **`INADDR_LOOPBACK`**).

The BeOS does not currently implement global binding. When you bind to **`INADDR_ANY`**, the `bind()` function binds to the first available interface (where "availability" means the address/port combination is currently unbound). Internet interfaces are considered before the loopback interface. If you want to bind to all interfaces, you have to create a separate socket for each. An example of this is given later.

- `sin_zero` is padding. To be safe, you should fill it with zeros.

The *size* argument is the size, in bytes, of the second argument.

If the `bind()` call is successful, the *interface* argument is set to contain the actual address that was used. If the socket can't be bound, the function returns a negative value, and sets the global **`errno`** to **`EABDF`** if the *socket* argument is invalid; for all other errors, **`errno`** is set to -1.

The following example shows a typical use of the `bind()` function. The example uses the fictitious `gethostaddr()` function that's defined in the description of the **`gethostname()`** function on page 23.

```
struct sockaddr_in sa;
int sock;
long host_addr;
```

```
/* Create the socket. */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
   /* error */
}

/* Set the address format for the imminent bind. */
sa.sin_family = AF_INET;

/* We'll choose an arbitrary port number translated to network byte order. */
sa.sin_port = htonl(2125);

/* Get the address of the local machine. If the address can't
* be found (the function looks it up based on the host name),
* then we use address INADDR_ANY.
*/
if ((host_addr = (ulong)gethostaddr()) == -1) {
   host_addr = INADDR_ANY;
}
sa.sin_addr.s_addr = host_addr;

/* Clear sin_zero. */
memset(sa.sin_zero, 0, sizeof(sa.sin_zero));

/* Bind the socket. */
if (bind(sock, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
   /* error */
}
```

As mentioned earlier, the bind-to-all-interfaces convention (by asking to bind to address 0) isn't currently implemented. Thus, if the `gethostaddr()` call fails in the example, the socket will be bound to the first address by which the local computer is known.

But let's say that you really do want to bind to all interfaces. To do this, you have to create separate sockets for each interface, then call `bind()` on each one. In the example below we create a series of sockets and then bind each one to an interface that specifies address 0. In doing this, we depend on the "first *available* interface" rule to find the next interface for us. Keep in mind that a successful `bind()` rewrites the contents of the **`sockaddr`** argument (most importantly, it resets the 0 address component). Thus, we have to reinitialize the structure each time through the loop:

```
/* Declare an array of sockets. */
#define MAXSOCKETS
int socks[MAXSOCKETS];
int sockN;
int bind_res;

struct sockaddr_in sock_addr;
```

```
for (sockN = 0; sockN < MAXSOCKETS; sockN++)
{
   (socks[sockN] = socket(AF_INET, SOCK_STREAM, 0));
   if (socks[sktr] < 0) {
      perror("socket");
      goto sock_error;
   }

   /* Initialize the structure. */
   sa.sin_family = AF_INET;
   sa.sin_port = htonl(2125);
   sa.sin_addr.s_addr = 0;
   memset(sa.sin_zero,0,sizeof(sa.sin_zero));

   bind_res = bind(socks[sockN],
               (struct sockaddr *)&sa,
               sizeof(sa));

   /* A bind error means we've run out of addresses. */
   if (bind_res < 0) {
      closesocket(socks[sockN--]);
      break;
   }
}

/* Use the bound socket (listen, accept, recv/send). */
...

sock_error:
   for (;sockN >=0 sockN--)
      closesocket(socks[sockN]);
```

To ask a socket about the address and port to which it is bound you use the **getsockname()** function, described later in this section.

RETURN CODES

Upon failure, `bind()` returns a negative value and sets **errno** to...

- **EABDF.** The *socket* argument is invalid.
- **B_ERROR**. All other errors.

---

connect()

int **connect(**int *socket*, const struct sockaddr **remote_interface*, int *remote_size***)**

The meaning of the `connect()` function depends on the type of socket that's passed as the first argument:

- If it's a stream client, then `connect()` attempts to form a connection to the socket that's specified by *remote_interface*. The remote socket must be a bound stream listener. A client socket can only be connected to one listener at a time. Note that you can't call `connect()` on a stream listener.
- If it's a datagram socket (either a sender or a receiver), `connect()` simply caches the *remote_interface* information in anticipation of subsequent **`send()`** and **`recv()`** calls. By using `connect()`, a datagram avoids the fuss of filling in the remote information that's needed by the "normal" datagram message functions, **`sendto()`** and **`recvfrom()`**. Note that a datagram may only call **`send()`** and **`recv()`** if it has first called `connect()`.

The *remote_interface* argument is a pointer to a **`sockaddr_in`** structure cast as a **`sockaddr`** pointer. The *remote_size* value gives the size of *remote_interface*. See the **`bind()`** function for a description of the **`sockaddr_in`** structure.

Currently, you can't disconnect a connected socket. If you want to connect to a different listener, or reset a datagram's interface information, you have to close the socket and start over.

When you attempt to `connect()` a stream client, the listener must respond with an **`accept()`** call. Having gone through this dance, the two sockets can then pass messages to each other through complementary **`send()`** and **`recv()`** calls. If the listener doesn't respond immediately to a client's attempt to connect, the client's `connect()` call will block. If the listener doesn't respond within (about) a minute, the connection will time out. If the listener's acceptance queue is full, the client will be refused and `connect()` will return immediately.

If the socket is in no-block mode (as set through **`setsockopt()`**), and blocking would occur otherwise, connect() returns immediately with a result of **`EWOULDBLOCK`**.

RETURN CODES

Upon failure, `connect()` returns a negative number and sets **`errno`** to...

- **`EWOULDBLOCK.`**,The connection attempt would block.
- **`EISCONN.`**,The socket is already connected.
- **`ECONNREFUSED.`**,The listener rejected the connection.
- **`ETIMEDOUT.`**,The connection attempt timed out.
- **`ENETUNREACH.`**,The client can't get to the network.
- **`EBADF.`**,The *socket* argument is invalid.
- **`B_ERROR`**. All other errors.

---

getpeername() , getsockname()

int **getpeername(**int *socket*, struct sockaddr **interface*, int **size***)*
int **getsockname(int** *socket, struct sockaddr *interface, int *size***)**

`getsockname()` returns, by reference in *interface*, a **sockaddr_in** structure that contains the interface information for the bound socket given by *socket*.

`getpeername()` returns, in the structure pointed to by the *interface* parameter, a **sockaddr_in** structure that describes the remote interface to which the *socket* is connected.

In both cases, the **size* argument gives the size of the *interface* structure; **size* is reset, on the way out, to the size of the interface argument as it's passed back. Note that the **sockaddr_in** pointer that you pass as the second argument must be cast as a pointer to a **sockaddr** structure:

```
struct sockaddr_in interface;
int size = sizeof(interface);

/* We'll assume "sock" is a valid socket token. */
if (getsockname(sock, (struct sockaddr*)&interface, &size) < 0)
   /* error */
```

RETURN CODES

Upon failure, `getsockname()` and `getpeername()` return negative numbers and set **errno** to...

- **EINVAL.**,The **size* value (going in) wasn't big enough.
- **EBADF.**,The *socket* argument is invalid.
- **B_ERROR**. All other errors.

---

listen() , accept()

int **listen(**int *socket*, int *acceptance_count***)**
int **accept(**int *socket*, struct sockaddr **client_interface*, int **client_size***)*

After you've bound a stream listener socket to an interface (through **bind()**), you then tell the socket to start listening for clients that are trying to connect. You then pass the socket to `accept()` which blocks until a client connects to the listener (the client does this by calling **connect()**, passing it a description of the interface to which the listener is bound).

When `accept()` returns, the value that it returns directly is a new socket token; this socket token represents an "accept" socket that was created as a proxy (on the local machine) for the client. To receive a message from the client, or to send a message to the client, the listener must pass the accept socket to the respective stream messaging functions, **recv()** and **send()**.

A listener only needs to invoke `listen()` once; however, it can accept more than one client at a time. Often, a listener will spawn an "accept" thread that loops over the `accept()` call.

Only stream listeners need to invoke `listen()` and `accept()`. None of the other socket types (enumerated in the **socket()** description) need to call these functions.

`listen()` takes two arguments: The first is the socket that you want to have start listening. The second is the length of the listener's "acceptance count." This is the number of clients that the listener is willing to accept at a time. If too many clients try to connect at the same time, the excess clients will be refused—the connection isn't automatically retried later.

After the listener starts listening, it must process the client connections within a certain amount of time, or the connection attempts will time out.

If `listen()` succeeds, the function returns 0; otherwise it returns a negative result and sets the global **errno** to a descriptive constant. Currently, the only **errno** value that `listen()` uses, other than -1, is **EBADF**, which means the socket argument is invalid.

The arguments to `accept()` are the socket token of the listener (*socket*), a pointer to a **sockaddr_in** structure cast as a **sockaddr** structure (*client_interface*), and a pointer to an integer that gives the size of the *client_interface* argument (*client_size*).

The *client_interface* structure returns interface information (IP address and port number) of the client that's attempting to connect. See the **bind()** function for an examination of the **sockaddr_in** structure.

The **client_size* argument is reset to give the size of *client_interface* as it's passed back by the function.

The value that `accept()` returns directly is a token that represents the accept socket. After checking the token value (where a negative result indicates an error), you must cache the token so you can use it in subsequent **send()** and **recv()** calls.

When you're done talking to the client, remember to call **closesocket()** on the accept socket that `accept()` returned. This frees a slot in the listener's acceptance queue, allowing a possibly frustrated client to connect to the listener.

RETURN CODES

Upon failure, `listen()` and `accept()` return < 0 and set **errno** to...

- **EBADF.**, The *socket* argument is invalid.
- **EINVAL. (accept() only)** The listener socket isn't bound.
- **EWOULDBLOCK. (accept() only)** The acceptance queue is full.
- **B_ERROR**. All other errors.

---

select()

int **select(**int *socket_range*, struct fd_set **read_bits*, struct fd_set **write_bits*,
struct fd_set **exception_bits*, struct timeval **timeout***)**

The `select()` function returns information about selected sockets. The *socket_range* argument tells the function how many sockets to check: It checks socket numbers up to (*socket_range* - 1). Traditionally, the *socket_range* argument is set to 32.

The **`fd_set`** structure that types the next three arguments is a 32-bit mask that encodes the sockets that you're interested in; this refines the range of sockets that was specified in the first argument. You should use the `FD_OP()` macros to manipulate the structures that you pass in:

- `FD_ZERO(`*set*`)` clears the mask given by *set*.
- `FD_SET(`*socket*, *set*`)` adds a socket to the mask.
- `FD_CLEAR(`*socket*, *set*`)` clears a socket from the mask.
- `FD_ISSET(`*socket*, *set*`)` returns non-zero if the given socket is already in the mask.

The function passes socket information back to you by resetting the three **`fd_set`** arguments. The arguments themselves represent the types of information that you can check:

- *read_bits* tells you if a socket is "ready to read." In other words, it tells you if a socket has a in-coming message waiting to be read.
- *write_bits* tells you if a socket is "ready to write."
- *exception_bits* tells you if there's an exception pending on the socket.

Currently, only *read_bits* is implemented. You should pass **`NULL`** as the *write_bits* and *exception_bits* arguments.

`select()` doesn't return until at least one of the **`fd_set`**-specified sockets is ready for one of the requested operations. To avoid blocking forever, you can provide a time limit in the final argument, passed as a **`timeval`** structure.

In the following example, we check if a given datagram socket has a message waiting to be read. The `select()` times out after two seconds:

```
bool can_read_datagram(int s)
{
   struct timeval tv;
   struct fd_set fds;

   tv.tv_sec = 2;
   tv.tv_usec = 0;

   /* Initialize (clear) the socket mask. */
```

```
    FD_ZERO(&fds);

    /* Set the socket in the mask. */
    FD_SET(s, &fds);
    select(32, &fds, NULL, NULL, &tv);

    /* If the socket is still set, then it's ready to read. */
    return FD_ISSET(s, &fds);
}
```

RETURN CODES

If `select()` fails, it returns -1; if the function times out, it returns 0. Otherwise (i.e. if *any* of the selected sockets was found to be ready) it returns 1.

---

send() , recv()

ssize_t **send(**int *socket*, const void **buf*, size_t *size*, int *flags***)**
ssize_t **recv(**int *socket*, void **buf*, size_t *size*, int *flags***)**

These functions are used to send data to a remote socket, and to receive data that was sent by a remote socket. `send()` and `recv()` calls must be complementary: after socket A sends to socket B, socket B needs to call `recv()` to pick up the data that A sent. `send()` sends its data and returns immediately. `recv()` will block until it has some data to return.

The `send()` and `recv()` functions can be called by stream or datagram sockets. However, there are some differences between the way the functions work when used by these two types of socket:

- For a stream listener and a stream client to transmit messages, the listener must have previously called **`bind()`**, **`listen()`**, **`accept()`**, and the client must have called **`connect()`**. Having been properly connected, the two sockets can send and receive as if they were peers.
- For stream sockets, `send()` and `recv()` can both block: `send()` blocks if the amount of data that's sent overwhelms the receiver's ability to read it, and `recv()` blocks if there's no message waiting to be read. You can tell these functions to be non-blocking by setting the sending socket's no-block socket option (see **`setsockopt()`**).
- If you want to call `send()` or `recv()` through a datagram socket, you must first **`connect()`** the socket. In addition, a receiving datagram socket must also be bound to an interface (through **`bind()`**). See the **`connect()`** description for more information on what that function means to a datagram socket.
- Datagram sockets never block on `send()`, but they can block in a `recv()` call. As with stream sockets, you can set a datagram socket to be non-blocking (for the `recv()`, as well as for **`recvfrom()`**) through **`setsockopt()`**.

The Arguments

The arguments to `send()` and `recv()` are:

- *socket* is, for datagrams and stream client sockets, the local socket token. In other words, when a datagram or stream client wants to send or receive data, it passes its own socket token as the first argument. The recipient of a `send()`, or the sender of a `recv()` is, for these sockets, already known: It's the socket that's identified by the previous **`connect()`** call.
- For a stream listener, *socket* is the "accept socket" that was previously returned by an **`accept()`** call. A stream listener can send and receive data from more than one client at the same time (or, at least, in rapid succession).
- *buf* is a pointer to the data that's being sent, or is used to hold a copy of the data that was received.
- *size* is the allocated size of *buf*, in bytes.
- *flags* is currently unused. For now, set it to 0.

RETURN CODES

A successful `send()` returns the number of bytes that were sent; a successful `recv()` returns the number of bytes that were received. Upon failure, the functions return negative numbers and set **`errno`** to...

- **`EWOULDBLOCK`**. The call would block on a non-blocking socket.
- **`EINTR`**. The local socket was interrupted.
- **`ECONNRESET`**. The remote socket disappeared (`send()` only).
- **`ENOTCONN`**. The socket isn't connected.
- **`EBADF`**. The *socket* argument is invalid.
- **`EADDRINUSE`**. The interface is busy (datagram sockets only).
- **`B_ERROR`**. All other errors.

---

sendto() , recvfrom()

ssize_t **`sendto(`**int socket, const void **buf*, size_t *size*, int *flags*, struct sockaddr **to*, int *toLen***`)`**
ssize_t **`recvfrom(`**int socket, void **buf*, size_t *size*, int *flags*, struct sockaddr **from*, int **fromLen***`)`**

These functions are used by datagram sockets (only) to send and receive messages. The functions encode all the information that's needed to find the recipient or the sender of the

desired message, so you don't need to call **connect()** before invoking these functions. However, a datagram socket that wants to receive messages must first call **bind()** (in order to fix itself to an interface that can be specified in a remote socket's sendto() call).

The four initial arguments to these function are similar to those for **send()** and **recv()**; the additional arguments are the interface specifications:

- For sendto(), the *to* argument is a **sockaddr_in** structure pointer (cast as a pointer to a **sockaddr** structure) that specifies the interface of the remote socket that you're sending to. The *toLen* argument is the size of the *to* argument.
- For recvfrom(), the *from* argument returns the interface for the remote socket that sent the message that recvfrom() received. **fromLen* is set to the size of the *from* structure. As always, the interface structure is a **sockaddr_in** cast as a pointer to a **sockaddr**.

sendto() never blocks. **recvfrom()**, on the other hand, will block until a message arrives, unless you set the socket to be non-blocking through the **setsockopt()** function.

You can broadcast a message to all interfaces that can be found by setting sendto()'s target address to **INADDR_BROADCAST**.

As an alternative to these functions, you can call **connect()** on a datagram socket and then call **send()** and **recv()**. The **connect()** call caches the interface information provided in its arguments, and uses this information the subsequent **send()** and **recv()** calls to "fake" the analogous sendto() and recvfrom() invocations. For sending, the implication is obvious: The target of the **send()** is the interface supplied in the **connect()**. The implication for receiving bears description: when you **connect()** and then call **recv()** on a datagram socket, the socket will only accept messages from the interface given in the **connect()** call.

You can mix sendto()/recvfrom() calls with **send()**/**recv()**. In other words, connecting a datagram socket doesn't prevent you from calling sendto() and recvfrom().

RETURN CODES

A successful sendto() returns the number of bytes that were sent; a successful recvfrom() returns the number of bytes that were received. Upon failure, the functions return negative numbers and set **errno** to...

- **EWOULDBLOCK**. The call would block on a non-blocking socket.
- **EINTR**. The local socket was interrupted.
- **EBADF**. The *socket* argument is invalid.
- **EADDRNOTAVAIL**. The specified interface is unrecognized.
- **B_ERROR**. All other errors.

---

setsockopt()

int **setsockopt(**int *socket*, int *level*, int *option*, const void **data*, uint *size***)**

`setsockopt()` lets you set certain options that are associated with a socket. Currently, the Network Kit only recognizes one option: It lets you declare a socket to be blocking or non-blocking. A blocking socket will block in a **`recv()`** or **`recvfrom()`** call if there's no data to retrieve.

A blocking socket will block in a **`send()`** or **`sendto()`** call if the send would overrun the network's ability to keep up with the data.

A non-blocking socket returns immediately, even if it comes back empty-handed or is unable to send the data.

The function's arguments are:

- *socket* is the socket that you're attempting to affect.
- *level* is a constant that indicates where the option is enforced. Currently, *level* should always be **`SOL_SOCKET`**.
- *option* is a constant that represents the option you're interested in. The only option constant that does anything right now is **`SO_NONBLOCK`**. (Two other constants—**`SO_REUSEADDR`** and **`SO_DEBUG`**—are recognized, but they aren't currently implemented.)
- *data* points to a buffer that's used to toggle or otherwise inform the option. For the **`SO_NONBLOCK`** option (and other boolean options), you fill the buffer with zeroes if you want to turn the option off (the socket will block), and non-zeros if you want to turn it on (the socket won't block). In the case of a boolean option, a single byte of zero/non-zero will do.
- *size* is the size of the *data* buffer.

RETURN CODES

Upon failure, `setsockopt()` returns a negative number and sets **`errno`** to...

- **`EINTR`**. The local socket was interrupted.
- **`EBADF`**. The *socket* argument is invalid.
- **`ENOPROTOOPT`**. Unknown *option*.

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetDebug

Derived from: (none)
Declared in: be/net/NetDebug.h
Library: libnetapi.so
Allocation: Constructor only
Summary: *more...*

BNetDebug provides a few functions that let you turn on and off debug output, and dump raw data to stderr in a clean format.

All the member functions are static; instead of creating a BNetDebug object, call them like this:

```
BNetDebug::Print("Starting server...");
```

Debug output is off by default; call **BNetDebug::Enable()** to enable it.

Static Functions

Dump()

static void **Dump(**const char ***data*, size_t *size*, const char ***title***)** const

Dumps *size* bytes of raw *data* to stderr. The dump is prefaced by the given *title*. The text is only output if debug output is currently enabled.

Enable() , IsEnabled()

static void **Enable(**bool *enable***)**

static bool **IsEnabled(**void**)**

`Enable()` enables debug output if enable is true, otherwise it disables debug output.

`IsEnabled()` reports the current state of debug output.

Print()

static void **Print(**const char ***message***)**

Prints the specified message to stderr, if debug output is currently enabled.

The Be Book The Network Kit The Network Kit Index

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BNetDebug Function Summary

Function Summary

Function	Description
Dump()	Dumps raw data to stderr
Enable()	Enables or disables debug output
IsEnabled()	Reports whether or not debug output is currently enabled
Print()	Prints the specified debugging message

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The OpenGL Kit Table of Contents

- The OpenGL Kit
- BGLView

The Be Book **The OpenGL Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The OpenGL Kit: Master Index

A

Adapting OpenGL Sample Code	BGLView
AttachedToWindow()	BGLView

C

Constructor and Destructor	BGLView
CopyPixelsIn()	BGLView
CopyPixelsOut()	BGLView

D

DirectConnected()	BGLView
Draw()	BGLView

E

EmbeddedView()	BGLView
EnableDirectMode()	BGLView
ErrorCallback()	BGLView

F

FrameResized()	BGLView

G

BGLView & BDirectWindow	BGLView
BGLView	BGLView
BGLView()	BGLView
~BGLView()	BGLView
The Graphics Buffers	BGLView

H

height	BGLView
Hook Functions	BGLView

L

linesize	BGLView
LockGL()	BGLView

M

Member Functions	BGLView

O

The OpenGL Kit	The OpenGL Kit

The OpenGL Kit	The OpenGL Kit
OpenGL On BeOS	The OpenGL Kit

P

pointsize	BGLView

S

SwapBuffers()	BGLView

U

UnlockGL()	BGLView
use_line_mode	BGLView
use_smooth_mode	BGLView
use_stipple_mode	BGLView
Using OpenGL	BGLView
Using OpenGL	BGLView

W

width	BGLView

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BGLView

Derived from: public BView
Declared in: be/opengl/GLView.h
Library: libGL.so
more...

The BGLView class provides a means for rendering graphics using OpenGL calls in a view.

The Graphics Buffers

A BGLView automatically manages video buffers and interfaces to supported hardware acceleration chipsets; all you have to do is call the appropriate OpenGL calls to render your graphics.

The **frontbuffer** is the graphics buffer that is currently visible on the screen. A **backbuffer** is a graphics buffer that is located offscreen.

In a single-buffered context, drawing commands are performed in the frontbuffer. Drawing performed in single-buffered mode immediately appears on the screen.

In double-buffered contexts, drawing is performed in the backbuffer and is not visible onscreen until the `SwapBuffers()` function is called to swap the two buffers and refresh the screen image.

The BGLView class currently only supports double-buffered OpenGL contexts.

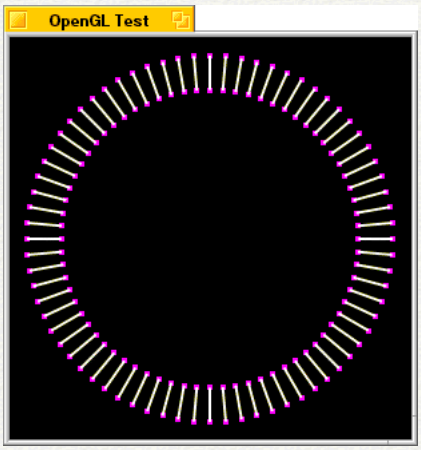
BGLView & BDirectWindow

The BGLView class provides a function, `DirectConnected()`, that your `BDirectWindow::DirectConnected()` function can call to handle the work of refreshing the OpenGL display.

Using OpenGL

Long-winded discussion of how to use OpenGL is well beyond the realm of what this book is intended to cover; for complete information on OpenGL, see the OpenGL web site at http://www.opengl.org, where you'll find complete documentation and sample code.

However, it's important to understand how OpenGL fits into the framework of a BeOS application. The example that follows will draw a pattern of lines around a central point, as seen in the picture below.



This code has been structured to make it relatively easy to port sample programs from the OpenGL web site; however, most of those samples use GLUT features, which aren't available yet in the BeOS implementation of OpenGL. In particular, most of the samples on the OpenGL web site use GLUT functions to handle user interface of some form. You'll have to add code for this yourself.

The complete source code and project file can be found on the Be web site at <<<insert URL here>>>.

The first thing that's needed, as always, is an application object, which we establish as follows:

```
class SampleGLApp : public BApplication {
   public:
                     SampleGLApp();
      private:
         SampleGLWindow      theWindow;
};
```

The SampleGLApp class has a constructor and a private pointer to the application's window. The constructor looks like this:

```
SampleGLApp::SampleGLApp()
         : BApplication("application/x-vnd.Be-GLSample") {
   BRect windowRect;
```

```
   uint32 type;

   // Set type to the appropriate value for the
   // sample program you're working with.

   type = BGL_RGB|BGL_DOUBLE;

   windowRect.Set(50,50,350,350);
   theWindow = new SampleGLWindow(windowRect, type);
}
```

The first thing the constructor here does is set the variable type to describe the context we need. In this example, we want an RGB context with double-buffering on, so we specify **BGL_RGB** and **BGL_DOUBLE**. Then we create the window using the new function.

The SampleGLWindow class is almost as simple:

```
class SampleGLWindow : public BWindow {
   public:
                  SampleGLWindow(BRect frame, uint32 type);
      virtual bool   QuitRequested();

   private:
      SampleGLView   *theView;
};
```

The constructor accepts a frame rectangle for the window and the OpenGL context type parameter that will be passed to SampleGLView's constructor. As always, **QuitRequested()** is overridden to post a **B_QUIT_REQUESTED** message to the application and return **true**. A pointer to the SampleGLView object is maintained as well.

The constructor is fairly trivial:

```
SampleGLWindow::SampleGLWindow(BRect frame, uint32 type)
         : BWindow(frame, "OpenGL Test", B_TITLED_WINDOW, 0) {
   AddChild(theView=new SampleGLView(Bounds(), type));
   Show();
   theView->Render();
}
```

This code establishes the window, then creates the SampleGLView and adds it as a child of the window. Once that's done, the window is made visible by calling **Show()**. Finally, the SampleGLView's contents are drawn by calling the SampleGLView's **Render()** function.

The meat of this program is in the SampleGLView class, which follows:

```
class SampleGLView : public BGLView {
   public:
```

```
                  SampleGLView(BRect frame, uint32 type);
      virtual void   AttachedToWindow(void);
      virtual void   FrameResized(float newWidth, float newHeight);
      virtual void   ErrorCallback(GLenum which);

      void         Render(void);

   private:
      void         gInit(void);
      void         gDraw(void);
      void         gReshape(int width, int height);

      float         width;
      float         height;
};
```

The SampleGLView class implements the constructor and reimplements three of the functions of the BGLView class: **`AttachedToWindow()`**, **`FrameResized()`**, and **`ErrorCallback()`**. An additional public method, `Render()`, will contain the actual code for drawing the contents of the view.

In addition, there are three private methods that will contain the actual OpenGL calls for initializing, drawing, and resizing the BGLView's contents and a pair of values to represent the width and height of the BGLView.

The constructor is very simple:

```
SampleGLView::SampleGLView(BRect frame, uint32 type)
         : BGLView(frame, "SampleGLView", B_FOLLOW_ALL_SIDES, 0,
            type) {
   width = frame.right-frame.left;
   height = frame.bottom-frame.top;
}
```

For the most part, the constructor defers to the BGLView constructor, setting the resizingMode to **`B_FOLLOW_ALL_SIDES`** and the OpenGL context type to the value specified.

The only addition is that the **`width`** and **`height`** of the view are cached, based upon the frame rectangle specified. This is done because we'll need that information when the view is attached to the window, and the BGLView class doesn't include **`Width()`** and **`Height()`** functions.

The **`AttachedToWindow()`** function, which is called when the SampleGLView is attached to its parent window, looks like this:

```
void SampleGLView::AttachedToWindow(void) {
   LockGL();
   BGLView::AttachedToWindow();
   gInit();
   gReshape(width, height);
```

```
    UnlockGL();
}
```

This performs the initialization of the OpenGL context. First, `LockGL()` is called to lock the context and let the OpenGL Kit know which view should be targeted by future OpenGL calls. Then the inherited version of `AttachedToWindow()` is called to let BGLView set up the view normally.

Once that's done, the `gInit()` and `gReshape()` functions are called. `gInit()`, as we'll see shortly, is responsible for initializing the context. `gReshape()` is called to configure the OpenGL coordinate system for the BGLView given the current **`width`** and **`height`** of the view.

Finally, `UnlockGL()` is called to release the OpenGL context for the SampleGLView and to indicate that we're done using the context for the time being.

The `FrameResized()` function is called automatically whenever the SampleGLView is resized:

```
void SampleGLView::FrameResized(float newWidth, float newHeight) {
    LockGL();
    BGLView::FrameResized(width, height);
    width = newWidth;
    height = newHeight;

    gReshape(width,height);

    UnlockGL();
    Render();
}
```

As always, this function begins by locking the OpenGL context. It then calls the inherited version of `FrameResized()` to let BGLView perform whatever tasks it may need to do.

The new width and height of the view are saved in the **`width`** and **`height`** variables, then the `gReshape()` function is called to adjust the OpenGL context given the new size of the view.

Finally, the context is unlocked, and `Render()` is called to redraw the view's contents at the new size.

Although the default `ErrorCallback()` function provided by BGLView would be acceptable, we include one of our own anyway:

```
void SampleGLView::ErrorCallback(GLenum whichError) {
    fprintf(stderr, "Unexpected error occured (%d):n", whichError);
    fprintf(stderr, " %sn", gluErrorString(whichError));
}
```

Note the use of the **`gluErrorString()`** OpenGL function to obtain an appropriate error message for the error code. You can use this function to avoid displaying error messages for errors that are acceptable or expected.

The `gInit()` function sets up the OpenGL context and initializes variables that will be used later:

```
void SampleGLView::gInit(void) {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glLineStipple(1, 0xF0E0);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE);
   use_stipple_mode = GL_FALSE;
   use_smooth_mode = GL_TRUE;
   linesize = 2;
   pointsize = 4;
}
```

Briefly, this sets the clear color (the background color) of the view to black, configures the pattern for stippled lines and the blending function to be used when blending is enabled. It also selects not to use stippled lines (you can change this by setting `use_stipple_mode` to **`GL_TRUE`**) and to use anti-aliasing when drawing the lines (you can change this by setting **`use_smooth_mode`** to **`GL_FALSE`**). It also chooses to use 2 pixel wide lines, and 4 pixel wide points.

This function doesn't call **`LockGL()`** and **`UnlockGL()`**, so they must be called by the calling function (and if you look at the **`AttachedToWindow()`** code above, you'll see that this is the case).

There are some global variables used by this program (some of them accessed in the above code), so lets' take a quick look at those:

```
GLenum use_stipple_mode;      // GL_TRUE to use dashed lines
GLenum use_smooth_mode;         // GL_TRUE to use anti-aliased lines
GLint linesize;               // Line width
GLint pointsize;            // Point diameter

float pntA[3] = {
   -160.0, 0.0, 0.0
};
float pntB[3] = {
   -130.0, 0.0, 0.0
};
```

The varaibles **`use_stipple_mode`** , **`use_smooth_mode`** , **`linesize`** , and **`pointsize`** are discussed in the `gInit()` function above. The two float arrays define points in three-dimensional space. These points will be used as the endpoints of the lines drawn by the `gDraw()` function.

The `gDraw()` function does the actual drawing into the OpenGL context:

```
void SampleGLView::gDraw(void) {
   GLint i;
```

```
   glClear(GL_COLOR_BUFFER_BIT);
   glLineWidth(linesize);

   if (use_stipple_mode) {
      glEnable(GL_LINE_STIPPLE);
   } else {
      glDisable(GL_LINE_STIPPLE);
   }

   if (use_smooth_mode) {
      glEnable(GL_LINE_SMOOTH);
      glEnable(GL_BLEND);
   } else {
      glDisable(GL_LINE_SMOOTH);
      glDisable(GL_BLEND);
   }

   glPushMatrix();

   for (i = 0; i < 360; i += 5) {
      glRotatef(5.0, 0,0,1);         // Rotate right 5 degrees
      glColor3f(1.0, 1.0, 0.0);      // Set color for line
      glBegin(GL_LINE_STRIP);         // And create the line
         glVertex3fv(pntA);
         glVertex3fv(pntB);
      glEnd();

      glPointSize(pointsize);         // Set size for point
      glColor3f(0.0, 1.0, 0.0);      // Set color for point
      glBegin(GL_POINTS);
         glVertex3fv(pntA);         // Draw point at one end
         glVertex3fv(pntB);         // Draw point at other end
      glEnd();
   }

   glPopMatrix();                  // Done with matrix
}
```

Without getting too deeply-involved in OpenGL specifics, this code begins by clearing the context's buffer and setting the line width. It then enables the features selected by the **`use_stipple_mode`** and **`use_line_mode`** variables.

Once that's done, it establishes a matrix to be used for rotating the lines and draws the lines with points at each end, drawing one every five degrees in a 360-degree circle around the center of the window. After drawing all the lines, the matrix is destroyed and the function returns.

The `gReshape()` function handles adjusting the OpenGL context's coordinate system and viewport when the SampleGLView is first created, and whenever the view is resized:

```
void SampleGLView::gReshape(int width, int height) {
   glViewport(0, 0, width, height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(-175, 175, -175, 175);
   glMatrixMode(GL_MODELVIEW);
}
```

This code simply sets the viewport's coordinate system to reflect the new width and height of the view, and establishes a projection matrix such that no matter what the size and shape of the window, the center of the window is considered to be (0,0) and the window is 300 units wide and 300 units tall. This lets the rendering code draw without having to worry about scaling; OpenGL handles it for us.

The details of how this works are, again, beyond the scope of this chapter.

Finally, the `Render()` function is the high-level function used to actually update the contents of the SampleGLView whenever we wish to redraw it:

```
void SampleGLView::Render(void) {
   LockGL();
   gDraw();
   SwapBuffers();
   UnlockGL();
}
```

`LockGL()` is called to lock the context before calling `gDraw()` to do the actual OpenGL calls to draw the view. Then the **`SwapBuffers()`** function is called to swap the backbuffer that was just drawn to the screen, and the context is unlocked.

Adapting OpenGL Sample Code

The program described above can easily be adapted to be used with other sample code from the OpenGL web site. First, replace the code in the `gInit()`, `gDraw()`, and `gReshape()` functions with the code from the `Init()`, `Draw()`, and `Reshape()` functions in the sample code (some of the sample programs give these functions slightly different names).

Keep in mind that the current implementation of OpenGL under BeOS doesn't support single-buffered graphics, so you'll need to make whatever adjustments are necessary to use double-buffering.

Once these functions have been implemented, copy any global variables from the sample program into your project. Finally, in the SampleGLApp constructor, fix the OpenGL context type and window size information to match that used by the sample program.

You may also wish to implement code to handle user interface to let you configure the sample program; that's beyond the scope of this chapter—see the Interface Kit chapter of the Be Developer's Guide for further information on handling user input.

Hook Functions

ErrorCallback()

Can be implemented to handle OpenGL errors.

Constructor and Destructor

BGLView()

```
BGLView(BRect frame, const char *name,
    int32 resizingMode,
    int32 flags, int32 type)
```

Initializes the view, then creates a new OpenGL drawing context and attaches it to the view. The *type* argument specifies the OpenGL type specification for the view:

BGL_RGB	Use RGB graphics instead of indexed color (8-bit). This is the default if beither **BGL_RGB** nor **BGL_INDEX** is specified.
BGL_INDEX	Use indexed color (8-bit graphics). **Not currently supported.**
BGL_SINGLE	Use single-buffering; all rendering is done directly to the display. This is not currently supported by the BeOS implementation of OpenGL. This is the default if neither **BGL_SINGLE** nor **BGL_DOUBLE** is specified.
BGL_DOUBLE	Use double-buffered graphics. All rendering is done to an offscreen buffer and only becomes visible when the **SwapBuffers()** function is called.
BGL_ACCUM	Requests that the view have an accumulation buffer.
BGL_ALPHA	Requests that the view's color buffer include an alpha component.
BGL_DEPTH	Requests that the view have a depth buffer.
BGL_STENCIL	Requests that the view have a stencil buffer.

~BGLView()

```
virtual ~BGLView()
```

Disposes of the OpenGL context for the view.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow**(void)

Calls the inherited version of `AttachedToWindow()` and sets the view color to **B_TRANSPARENT_32_BIT** (this improves performance by preventing the Application Server from erasing the view, since OpenGL takes over responsibility for drawing into the view).

CopyPixelsIn() , CopyPixelsOut()

status_t **CopyPixelsIn**(BBitmap **source*, BPoint *dest*)

status_t **CopyPixelsOut**(BPoint *source*, BBitmap **dest*)

These functions copy pixel data into and out of the OpenGL draw buffer for the context.

`CopyPixelsIn()` copies the entire contents of the *source* BBitmap into the OpenGL context, offset such that the top-left corner of the BBitmap is drawn at the point *dest* in the OpenGL buffer.

`CopyPixelsOut()` copies from the OpenGL draw buffer into the specified BBitmap. The area copied is the size of the *dest* bitmap and contains all data from the specified *source* point to the bottom-right corner of the buffer.

If the source is larger than the destination, it's clipped at the bottom and right edges to fit; no scaling is performed Also, the OpenGL context and the BBitmap must be in the same color space.

RETURN CODES

- B_OK. The data was copied without error.
- **B_BAD_VALUE**. The current draw buffer is not valid, or the destination buffer's width or height is less than or equal to zero.
- **B_BAD_TYPE**. The source and destination are in different color spaces.

CopyPixelsOut() see CopyPixelsIn()

DirectConnected()

void **DirectConnected**(direct_buffer_info **directInfo*)

If the BGLView is in a BDirectWindow, you should call this from your **BDirectWindow::DirectConnected()** function to let OpenGL update the window properly.

Draw()

virtual void **Draw(**BRect *updateRect***)**

Refreshes the contents of the BGLView by copying the frontbuffer to the screen.

If the view's color space is eight bits deep and the **GL_DITHER** OpenGL option is enabled, the display is dithered.

EmbeddedView()

BView ***EmbeddedView(**void**)**

Returns a pointer to an embedded view that encompasses the current OpenGL drawing buffer, as defined by OpenGL, for the BGLView. If the view is single-buffered, this will be the frontbuffer, and if the view is double-buffered, the embedded view will encompass the backbuffer.

`EmbeddedView()` returns **NULL** if, for any reason, BView functions can't be used in the GL buffer. Starting with BeOS R4, this function always returns **NULL**, as the new, high-performance implementation of OpenGL does not support tinkering with the view.

EnableDirectMode()

void **EnableDirectMode(**bool *enabled***)**

Call this function to tell the BGLView whether or not it should render in direct mode. If you're using a BDirectWindow and want video refreshes to be done through the direct window method, pass **true** for *enabled*. If you don't want to use the direct window method, pass **false**.

ErrorCallback()

virtual void **ErrorCallback(**GLenum *errorCode***)**

Called when an OpenGL error occurs. By default, this function invokes the debugger with an error message reading "GL: Error code $xxxx." You can (and probably should) reimplement this function to cope with errors more gracefully.

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Calls the inherited version of `FrameResized()`, releases tables that need to be recalculated, and resizes the OpenGL buffers.

You can augment this function to perform other necessary tasks, such as adjusting your BGLView's coordinate system.

LockGL() , UnlockGL()

void **LockGL(**void**)**
void **UnlockGL(**void**)**

These functions lock and unlock the OpenGL context. You must lock the context before issuing any OpenGL commands, and unlock it when you're done—this is how OpenGL knows which context the drawing commands are intended for, since OpenGL itself isn't encapsulated within the BGLView class. For example:

```
LockGL();            /* lock the OpenGL context
glEnable(GL_DITHER);         /* turn on dithering support */
UnlockGL();
```

Failing to lock and unlock the context appropriately will result in unpredictable behavior and may cause your application to crash.

SwapBuffers()

void **SwapBuffers(**void**)**
void **SwapBuffers(**bool *vSync***)**

Swaps the front buffer and back buffer, then redraws the contents of the BGLView.

This function has no effect if the view is single-buffered.

If *vSync* is specified and is **true**, the swap is synchronized with vertical blanking.

UnlockGL() see LockGL()

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

BGLView Summary

The OpenGL Kit Table of Contents The OpenGL Kit Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

The OpenGL Kit

The OpenGL Kit provides an interface between your BeOS application and the OpenGL graphics library, which is provided with BeOS. The one class in the OpenGL Kit, BGLView, lets you display graphics rendered using OpenGL on a computer running the BeOS.

Before Release 4.5, the OpenGL Kit also had a BGLScreen class. BGLScreen functionality has been subsumed by BGLView. Old BGLScreen code will still run, but new OpenGL Kit code must use BGLView objects only.

The BGLView class is used to create a view within a window that contains OpenGL-rendered data. Derived from BView, it adds functions for locking and unlocking the OpenGL context associated with the view, as well as for copying pixel data into and out of the graphics buffer, and swapping the front and back buffers.

If you want to use OpenGL graphics in a BDirectWindow, create your BDirectWindow and attach a BGLView to it.

OpenGL On BeOS

BeOS has included an OpenGL implementation since the first Preview Release for PowerPC processors. This implementation of OpenGL is complete, and GLU is supported as well. The optional AUX and GLUT libraries, however, aren't supported at this time.

Also, the current implementation of OpenGL on BeOS supports only 32-bit graphic buffers. Your BGLView or BGLScreen can be in any graphics mode you want, but the graphics buffer offscreen is always 32-bit.

Complete descriptions of the features and use of OpenGL are beyond the scope of this book; however, you can get full documentation of OpenGL as well as sample code at the OpenGL web site at http://www.opengl.org.

There are also some sample OpenGL programs that have already been ported to BeOS available for download on the Be web site; visit http://www.be.com/developers/topics/opengl.html.

The Be Book **The OpenGL Kit** **The OpenGL Kit Index**

The Be Book

,

...in lovely HTML...

for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Be Book Table of Contents

Drag and Drop

This is the reference document for constructing BMessage objects as used in BeOS' standard negotiated drag'n'drop actions. If you're reading it for the first time, you should probably read the entire thing. If you want to look up something and you're reading this with a browser, you can jump to any of the following:

1. Simple vs. Negotiated Drag'n'Drop

2. Basics of Drag and Drop

3. BMessage Formats in Negotiated Drag'n'Drop

3a. Protocol for the drag message

3b. Protocol for the Negotiation Message

3c. Protocol for the Data Message

4. Summary of the Negotiated Drag'n'Drop Protocol

Prerequisites

Drag and Drop makes use of a number of elements of the BeOS programming kit, and you'll need to be at least reasonably familiar with the following topics, in order to be comfortable reading this document:

- The **`BMessage`** class, since dragged objects are sent to their targets as **`BMessage`** objects. You should understand what a **`BMessage`** is, how to add data to a **`BMessage`** or extract data from it, how to send a **`BMessage`**, and where to put code to handle incoming **`BMessage`**s.
- The **`BView`** class. The drag and drop action is fundamentally a GUI operation, and **`BView`** objects are the most important objects in handling GUI events; subclasses of **`BView`** are where you will most often define code to both initiate a drag, and handle a drop.
- The Translation Kit, if you wish to make use of the BeOS' built-in translation abilities to (for example) drag a JPEG image from an application, to another application that only understands raw bitmaps.

Simple vs. Negotiated Drag'n'Drop

Roughly speaking, you can do two different kinds of drag and drop operations; both appear the same to the user, but they differ in the complexity of the background communication between the source of the drag, and the drop target.

Simple Drag'n'Drop

A *simple* drag'n'drop is one where the action of dragging and dropping corresponds internally to a single **`BMessage`** being sent, from the origin of the drag to the drop target. This was the original drag'n'drop protocol for the BeOS, and is still highly useful in dragging and dropping which takes place within a single application, or within a tightly coupled set of applications which "know about" one another. For example, you could drag around drawing elements of a drawing program using this protocol; each drag operation would result in the drawing application sending a **`BMessage`** within itself, and that **`BMessage`** would contain the data necessary to indicate which drawing element had been moved, and to where. Using the built in drag'n'drop abilities of the BeOS in this manner will lessen your own workload, since many of the most basic aspects, such as visually dragging a bitmap around on the screen, are handled automatically.

Negotiated Drag'n'Drop

A *negotiated* drag and drop is used in a more general context, when the originator of a drag and the receiver (drop target) of

that drag do not necessarily know anything about one another. This more general drag'n'drop is necessary if we wish to make usage of the BeOS, particularly it's media-related applications, as intuitive to the user as possible. For example, the user should expect to be able to drag an image from one graphics program (say a viewer) to another graphics program (such as a paint program) with no difficulty. However, there are many different underlying data formats for representing images, and the two graphics programs will have to negotiate (with the aid of the Translation kit) for the most suitable or desirable format.

Note that the negotiated drag'n'drop protocol is not a "hard and fast" requirement; it is not enforced by the system, and could easily be modified; for example, you might need a particularly complex negotiation that takes more steps than defined in the standard. The major user of negotiated drag'n'drop you are likely to encounter right now is in Tracker, and we will talk about that in more detail later. Other applications may support negotiated drag'n'drop or may not. Also note that a simpler protocol was originally defined, and may still be used by older applications. It's easy to make your code handle this older protocol; see the later section on "Handling the Old Drag'n'Drop Protocol".

Basics of Drag and Drop

Let's assume the user is working with two applications, and is going to drag something from the first application (which we'll call the sender application) to the second application (which we'll call the receiver application.) In a nutshell, here are the steps that make up this entire process:

1. **Detection**: Code in a **`BView`** object of the sender application notices that the user has clicked on something of interest, and started dragging it. The monitoring code that notices this is usually spread out between the **`BView::MouseDown()`** and **`BView::MouseMoved()`** functions, and typically has to differentiate between the start of a real drag and drop, or a click that might (for example) select an object without dragging it, but that moves just one or two pixels because the user has a slightly jittery hand. The drag detection code will typically have to use **`BView::SetMouseEventMask()`** to temporarily enable full reporting of mouse movement (which is not normally reported to the application, for efficiency reasons).

2. **Initiation**: Once the sender application's detection code has decided a drag and drop action has been started, it initiates the drag and drop internally, by creating a **`BMessage`** object containing information about the nature of the object being dragged, and then passing that **`BMessage`** to the **`BView::DragMessage()`** function. **`BView::DragMessage()`** also determines how the object being dragged appears visually to the user; you can pass it a **`BBitmap`**, in which case it will drag a cool picture around the screen, or you can pass it a **`BRect`**, in which case it will drag a boring old rectangular outline. After calling **`DragMessage`**, the sender's code

should remember to delete its own copy of the **`BMessage`**. **Note**: The sender should not delete the **`BBitmap`** it passes to **`DragMessage`**; this will be done by the system

3. Drag: This is the easy part; once the sender application has initiated the drag, as described in the previous step, dragging a visual representation around the screen is handled by the system. The next time application code is invoked is on the . . .

4. Drop: The user releases the mouse button while the mouse is in a **`BView`** of the receiver application. This causes the **`MessageReceived()`** function of the recipient **`BView`** to be invoked, with the dragged **`BMessage`** as its received message. This is the first of three **`BMessage`** objects that can be sent as part of a single drag and drop action. We'll call this first message the *drag message*. In a simple drag'n'drop, the drag message can contain just about anything you want it to; since both the sender and the receiver "know about" one another, they presumably have a common private protocol, and the receiver can analyze the message and do whatever is required. In a negotiated drag'n'drop, the drag message does *not* typically carry the data that defines the object being dropped; for example, if you drop a block of text, the message that is dropped normally does not contain the text that was dragged. Instead, the drag message contains information about the different formats and methods by which the sender application may supply the data to the receiver application, and about which actions the receiver application can request of the sender application.

5. Negotiation [optional, used in negotiated drag'n'drop]: The receiver application replies to the drag message with a second message, which is dispatched back to the sender application using the **`BMessage::SendReply()`** function. We'll call this second

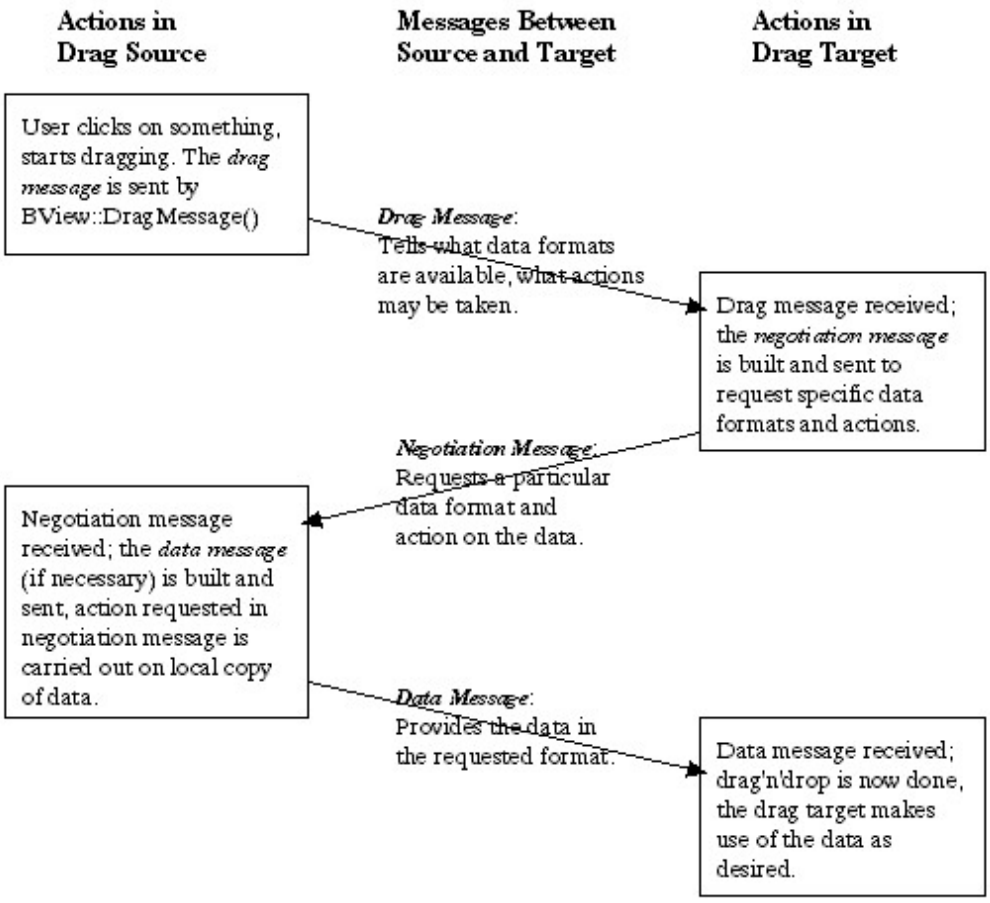
message the *negotiation message*. The receiver creates the negotiation message by looking through the options supplied by the sender in the drag message, choosing one or more, and including those option in the message so that the sender will know which of its possible replies is actually desired. The negotiation message also contains information about what the receiver would like the sender to do with the dragged data; for example, if you drag a file to the Trash, the negotiation message sent back "from" the Trash will include a request that the sender delete the dragged data.

6. **Data Transmission** [optional, used in negotiated drag'n'drop]: The sender examines the negotiation message sent back to it by the receiver, to find out which available data format the receiver wants the data in. The sender then packages up the data, and dispatches it off to the receiver in the *data* message. (Optionally, the sender can, at the request of the receiver, create a file through which it passes the data to the receiver, rather than passing the data directly in the **`BMessage`**. We'll talk about this when we discuss the details of the drop, negotiation, and data messages.)

7. **Completion** [optional, used in negotiated drag'n'drop]: The receiver gets the data message from the sender, extracts the desired data, and uses it in whatever way it wants.

This sounds more complex than it is. A picture is worth a thousand words, so here is a simplified schematic of the process:

Drag'n'Drop Schematic Description



Example: Simple Drag'n'Drop

Before going on to more complex matters, let's see just how simple "simple" drag'n'drop can be. The code below defines a very simple application, which lets one drag around a black square in a window. The application is both the sender (i.e., you click on the square in the application window to start the drag), and the receiver (you release the mouse button within the application window, to reposition the square). Most of the program code is "boilerplate", similar to what would be found in any application involving a **`BView`**. Those parts of the code pertaining specifically to the drag and drop are given in bold.

```
   /* Drag the black square around using drag'n'drop--a very
      simple-minded program. The DETECTION, INITIATIOIN, DRAG,
      and DROP steps correspond to those of the same name given
      in the overview section, "Basics of Drag and Drop". */

   #include <Application.h>
   #include <InterfaceKit.h>

   rgb_color black = {0, 0, 0, 64};
   const char *APP_SIGNATURE            = "application/x-vnd.Be-
MyDragnDrop";
   /* The constant below is arbitrary--the fact that is is
'drag' is
      meaningless. However, it MUST be chosen so as not to
conflict
      with system-wide BMessage 'what' values--see the BMessage
      documentation for details. */
   const uint32 MY_DRAG_ACTION = 'drag';

   class MyDragDropView : public BView {
      private:
         BPoint _button_click;
```

```
      BRect _SquareSides;

   public:
      MyDragDropView(BRect rect) : BView(rect, "",
                        B_FOLLOW_ALL_SIDES, B_WILL_DRAW) {
         _SquareSides = BRect(20, 20, 90, 90);
      }

      void Draw(BRect where) {
         SetHighColor(black);
         FillRect(_SquareSides);
      }

      void MouseDown( BPoint where ) {
         /* 1. DETECTION : Our detection code is pretty
simple-minded;
            if the user clicks in the black square, they're
starting a drag. */
         if (_SquareSides.Contains(where)) {
            // Turn on the event mask for all pointer events,
so we'll know when
            // the user lets go of the mouse button.
            SetMouseEventMask( B_POINTER_EVENTS, 0 );

            /* 2. INITIATION : Create a BMessage instance,
and pass it
               to BView::DragMessage() to start the drag. */
            BMessage *drag_message = new BMessage(
MY_DRAG_ACTION );
            /* Remember, in the drag message, where the drag
started */
            drag_message->AddPoint("click_location", where);
            DragMessage( drag_message, _SquareSides, this );
            /* 3. DRAG : this is handled by the user and
system, we
               don't write any code at all to do the
dragging. The
               next step will take place in the
MessageReceived() function. */
```

```
            /* DON'T FORGET TO DELETE THE MESSAGE AFTER
YOU'RE DONE*/
            delete drag_message;
         }
      }; /* end of MouseDown() */

      void MessageReceived(BMessage *msg) {
         /* 4. DROP : MessageReceived() can called for many
different
            messages. We're only interested in ones with a
'what'
            field of MY_DRAG_ACTION, as was created in the
MouseDown()
            function. */
         if (msg->what == MY_DRAG_ACTION) {
            BPoint clicked;
            /* We put "click_location" into the original
message, now
               we can get it out */
            msg->FindPoint("click_location", &clicked);
            BPoint whereto;
            /* The "_drop_point_" message field is
automatically
               inserted by the OS; is is the screen point the
               mouse was on when the drag ended. */
            msg->FindPoint("_drop_point_", &whereto);
            /* Convert "_drop_point_" to view coordinates. */
            whereto = ConvertFromScreen(whereto);
            /* Move the square by the same amount the mouse
moved
               in the drag. */
            _SquareSides.OffsetBy(whereto-clicked);
            /* Ensure the view is redrawn */
            Invalidate();
         }
      }; /* end of MessageReceived() */
   }; /* end of MyDragDropView class */
```

```
   class MyDragDropWindow : public BWindow {
      public:
         MyDragDropWindow(BRect frame)
            : BWindow(frame, "Drag'n'Drop Example",
B_TITLED_WINDOW, B_NOT_ZOOMABLE) {

            interior = new MyDragDropView(this->Bounds());
            AddChild(interior);
            Show();
         }

         bool QuitRequested() {
            be_app->PostMessage(B_QUIT_REQUESTED);
            return true;
         }
      private:
         BView *interior;
   }; /* end of MyDragDropWindow class */

   class MyDragDropApp : public BApplication {
      public:
         MyDragDropApp::MyDragDropApp() :
BApplication(APP_SIGNATURE) {
            BRect windowRect;
            windowRect.Set(50,50,349,399);
            new MyDragDropWindow(windowRect);
         }
      private:
         MyDragDropWindow   *theWindow;
   }; /* end of MyDragDropApp class */

   /* The "main" function creates and runs the application.*/
   int main(void) {
      MyDragDropApp *theApp;
      theApp = new(MyDragDropApp);
      theApp->Run();
      delete theApp;
   }
```

BMessage Formats in Negotiated Drag'n'Drop

With simple drag and drop, you are free to use **`BMessage`**s more or less as you wish; you simply define a **`what`** member constant which will uniquely identify your dropped message to the target, and put whatever data you wish into the body of the message. The receiver will examine the **`what`** member of the message and, assuming it understands it at all, should then know how to extract the data (if any) in the body of the message.

Negotiated drag and drop is different, since it assumes that the sender and receiver have no implicit knowledge of each other. In order for the two ends of the action to communicate effectively, they must agree on a standard protocol between them. This protocol takes the form of an agreed-up structure for each of the drag message, negotiation message, and data message, as described in the following sections.

Hint: There's a great little utility called *ViewIt*, which should be available (as part of a package of utilities called *Geb's Goodies*) on the usual download sites, such as www.bebits.com. You can drag anything onto *ViewIt*, and it will display the contents of the drag message.

Protocol for the Drag Message

The drag message is the one initially sent out by the sender application. It must be assigned a **`what`** member of **`B_SIMPLE_DATA`** by you, the programmer. The message fields in the drag message

can be split into those that are provided by you, and those that are filled in automatically by the system. Let's take a look at each of these.

Drag Message Fields Filled In by Your Code

The following drag message fields will need to be filled in by your code. Detailed descriptions of these fields are found below.

"**be:types**", "**be:filetypes**", and "**be:type_descriptions**": These fields indicates the formats the sender is willing to provide data in, and are filled in together.

"**be:actions**": A list of actions the sender is willing to perform on its data, at the request of the receiver.

"**be:clip_name**": A suggested name for the data being sent, which may be used at the option of the receiver. This field is optional.

"**be:originator**", "**be:originator_data**": used to preserve context information across asynchronous message sendings.

"**be:data**": This was used by the old [obsolete] drag'n'drop protocol, to contain the data being dragged. You don't need to use it with modern applications.

Drag Message Fields Filled In by the System

"**_drop_point_**", "**_drop_offset_**": The values in these fields are set automatically by the system; do not set them yourself. They give information about where the drop took place on the screen.

Detailed Description of the Drag Message Fields

"be:types", "be:filetypes", and "be:type_descriptions" Fields

The main purpose of negotiated drag'n'drop is to come up with an optimal compromise between the data formats the sender application can provide something in, and the data formats the receiver can accept that data in. Accordingly, the three most important message fields in the drag message are the **"be:types"** and **"be:filetypes"** fields; **"be:type_descriptions"** accompanies the "be:filetypes" field, so we'll describe it here also.. Each of these fields is filled with a list of string values.

Values for these fields will typically be obtained from the Translation Kit, by asking the Translation Kit which data formats it can provide the dragged data in, and filling the fields in with the information returned by the calls to the Translation Kit. (For example, if appropriate translators are installed on the system, the Translation Kit might be able to translate a bitmap into any of JPEG, GIF, PNG, or TIFF images.)

The specific meanings of these three fields are as follows:

- **"be:types"**: each value in this field indicates a data format the sender is willing to provide its sent data in. This is a string value (a MIME string, actually), but you won't normally need to know what the string actually is, since you'll obtain it from the Translation Kit, or from a predefined constant. If "be:types" consists of a single element whose value is equal to that in the global variable **`B_FILE_MIME_TYPE`**, then the sender application is not willing to provide its data in a BMessage; the receiver must accept the data through a file, in one of the formats specified in the "be:filetypes" field of

this message. If you want to indicate that your application is willing to send its data via either of a **`BMessage`** or a file, then add **`B_FILE_MIME_TYPE`** as the last element of "be:types", after all of the types which can be sent in a **`BMessage`**.

- **"be:filetypes"**: similar "be:types", but a values in this field indicate formats in which the sender is willing to provide its data *via a file*. This can make a lot of sense; for example, if what the user is dragging around is a 30 megabyte video clip, you probably don't want to pass this in a **`BMessage`**. In most cases, "be:filetypes" will be identical to "be:types". If your application isn't willing to pass data via a file, then it doesn't matter what (if anything) you put into "be:filetypes". **Important**: See the section below in "Passing Data Via a File" for further notes on this, if you plan to make use of it.

- **"be:type_descriptions"**: this is just a user-friendly description of the corresponding format in "be:filetypes". It may be displayed onscreen to the user when the user can choose between several different data formats during a file save operation. It is not used otherwise.

The "be:actions" Field

In negotiated drag'n'drop, the sender and receiver negotiate not only the format of the sent data, but also the action performed on that data. Should the data be copied from the sender to the receiver, moved from the sender to the receiver, or something else? The first part of this "actions negotiation" takes place in the drag message via the "be:actions" field.

"be:actions" contains a list of values (32-bit integers actually) which define the actions the sender is will to perform at the

request of the receiver. No action will actually be carried out until and unless the receiver requests it. The possible action requests are given by the following constants:

- **B_COPY_TARGET** : the sender is willing to provide a copy of the dragged object.
- **B_MOVE_TARGET** : The sender is willing to move the target over to the receiver; if the receiver requests this, then after the sender sends a copy of the data, it will delete its own copy.
- **B_LINK_TARGET** : The sender is willing to provide a link to the target. *[xxx what does this mean exactly? If the sender is the Tracker, I can understand it, but in Chris' document, B_LINK_TARGET is listed as a general action, not one that is Tracker-specific.]*
- **B_TRASH_TARGET** : The sender is willing to delete its data without even sending it. This is useful if, for example, you want your user to be able to drag something from the application to the Tracker's wastebasket icon, and have the object removed from the application as a result.

In addition to the above four "standard" actions, there are a few actions which can be carried out specifically when the Tracker is the sender application *[xxx does the Tracker have to be the receiver as well? If so, should we even document these actions?]*:

- **B_COPY_SELECTION_TO** : One or more elements (files, directories) are selected in a Tracker window; the Tracker is willing to copy these files to an **`entry_ref`** type destination,

which will be supplied in a "refs" field in the negotiation message, should the receiver application choose this action.

- **B_MOVE_SELECTION_TO** : One or more elements (files, directories) are selected in a Tracker window; the Tracker is willing to move these files to an **`entry_ref`** type destination, which will be supplied in a "refs" field in the negotiation message, should the receiver application choose this action.

[xxx Chris' document implies the Tracker can also make links due to a specific Tracker action, is there something like a B_LINK_SELECTION_TO?]

The "be:clip_name" Field

This is an easy one. If present (it doesn't have to be), it contains a string suggesting a name for the data which will be sent. This name may be used by the receiver; for example, if data is dragged onto the Tracker, the Tracker will try to use the value in "be:clip_name" as the basis for the name of the clipping file it creates from the data. However, the receiver is under no obligation to pay attention to "be:clip_name".

"be:originator" and "be:originator_data" Fields

Let's say you're a sender application, and you've asynchronously sent off a drag message. Then you receive a **`BMessage`**. Is it the reply to your original message? And if so, how do you access data about the original drag (such as where the mouse was clicked) you might need in order to complete the data message? Since the original drag message was sent asynchronously, your application went merrily on its way immediately after sending, and no longer

remembers what the heck it dragged off before. What do you do? This is where "be:originator" and "be:originator_data" come in.

- "**be:originator**" should be filled in with something that identifies your application to itself; it doesn't really matter what. We'll see how this is used in a second...
- "**be:originator_data**" can be filled in with data about the drag that you'll need to use later on in the drag'n'drop negotiation. This could be something as simple as the mouse position, or it could be a much more complex piece of information about the context of the application at the time the drag was started. It's up to you, and of course, you don't have to use "be:orginator_data" at all.

If you fill in "be:originator" and "be:originator_data" properly on all of your outgoing drag messages, then here is what your app can do when it receives a **`BMessage`** that *may* be a negotiation message responding to a previously sent (asynchronous) drag message:

1. Your app checks to ensure that the incoming message has the correct format for a negotiation message (we'll describe the format of negotiation messages in a bit). If it does, then go on...

2. Your app uses **`BMessage::IsReply()`** to check if the incoming message is a reply to a previous message.

3. If the incoming message is in fact a reply, your app uses **`BMessage::Previous()`** to obtain the original message; the message the incoming message was sent in reply to.

4. Now, your app can examine "be:originator" in the original message, to see if it recognizes that value as indicating a drag

message sent out by itself. If it does, then the incoming **`BMessage`** is in fact a negotiation message in response to your original drag message; your app can extract necessary context data from "be:originator_data", and go on to construct and send the data message.

The "be:data" Field

This field was used in the original drag'n'drop protocol to carry the dragged data. It is not used in the negotiated drag'n'drop protocol.

The "_drop_point_" Field

This field contains a **`BPoint`** giving the *screen* coordinate of the mouse cursor, when the drag was ended (i.e. when the mouse button was released and the data dropped.) It's added automatically by the system--you don't create or add to it yourself.

The "_drop_offset_" Field

Whether you have your application display a bitmap or a rectangular outline during a drag, the dragged area occupies a rectangle. (Though it may not appear so to the user, since with a bitmap, sections of the bitmap can be transparent). "**_drop_offset_**" gives as a **`BPoint`** the distance from the top left point of the dragged area to the position of the mouse cursor within that dragged area. *[xxx I think this is correct but just want to be sure.]*It's added automatically by the system--you don't create or add to it yourself.

Passing Data in a File

Using a **BMessage** to pass a large amount of data from one application to another may not be desirable; at the extreme, you may not have enough memory to accommodate the entire **BMessage**. If you need to pass large amounts of data in a drag'n'drop operation, you may prefer to do so via a file.

As far as passing data by a file goes, the sender application can indicate one of two things when sending the drag message to the receiver:

- The sender can offer to pass the data in a file, as well as via a **BMessage**. The receiver then has the option of accepting the data either through a **BMessage**, or through a file.
- The sender can state that it will only pass the data via a file. The receiver app, if it wants the data, had better agree.

In both cases, the sender's ability to send data via a file is indicated by a value in the drag message's "be:types" *field of* **B_FILE_MIME_TYPE**. If **B_FILE_MIME_TYPE** is the *first* entry in the "be:types" field, than the sender will pass data *only* in a file, and anything else in "be:types" will be ignored; if there are entries in "be:types" *before* a value of **B_FILE_MIME_TYPE**, then those entries are types with which the sender is prepared to send data *directly in* the data message.

If the sender has indicated, via the presence of a **B_FILE_MIME_TYPE** value in "be:types", that it is willing to pass data via a file, then the formats in which it can provide that file are listed in the "be:filetypes" field. If a value if **B_FILE_MIME_TYPE** is not present somewhere in the "be:types" field, then the sender is not able to pass data via a file, and any values in the "be:filetypes" field will be ignored.

Protocol for the Negotiation Message

When the receiver application receives the initial drag message from the sender application, it examines the data message in that message, to ascertain what actions the sender application can carry out, and how it can provide the desired data in the final drag message. Using this information, the receiver application formulates the negotiation message, which it sends back as a reply to the original drag message using the **`BMessage::SendReply()`** function.

A Negotiation Message is Characterized by its 'what' Member

Recall that the drag message constructed by the sender application contained (among other things) a "be:actions" message field, which indicated the actions the sender was willing to perform upon the dragged data. The allowable actions (at the time of this writing--further actions may be added in the future) are given by the following constants:

- **`B_COPY_TARGET`**
- **`B_MOVE_TARGET`**
- **`B_LINK_TARGET`**
- **`B_TRASH_TARGET`**
- **`B_MOVE_SELECTION_TO`** (for Tracker drags only.)
- **`B_COPY_SELECTION_TO`** (for Tracker drags only.)

When constructing the negotiation message, the receiver application will choose one of the actions listed in the data message's "be:types" message field (remember, not all allowable actions will necessarily be listed in this message field), and use that action as the value of the what member in the negotiation message. This will inform the sender application of which of the possible actions is desired by the receiver.

Message Fields in the Negotiation Message

In addition to the action contained in the `what` member, the negotiation message may also contain a number of message fields; exactly which message fields are defined depends somewhat on the requested action. The allowable message fields are:

- "**be:types**": One or more strings denoting the data formats the receiver is willing to accept the dragged data in, as part of the data message. These should be chosen from the "be:types" field of the drag message. If the "be:types" field of the negotiation message contains more than one value, then the first value is the format the receiver would prefer to get data in, but the sender may choose any of the formats, at its discretion. Of course, if the sender has indicated in the drag message that it will only pass data through a file, values in the negotiation message's "be:types" message field will be ignored when the negotiation message gets back to the sender.

- "**be:filetypes**": One or more strings denoting the data formats the receiver is willing to accept file data in, assuming of course that the sender indicated in its drag message that it was willing to provide the dragged data via a file. The values

in the negotiation message's "be:filetypes" field should be chosen from those provided in the drag message's "be:filetypes" field. It the negotiation message contains more than one value in "be:filetypes", then it would prefer to be given a file in the first listed format, but the sender is free to use any of the listed formats.

- "**directory**": If the sender decides to (or is forced to) pass data to the receiver via a file, the **`entry_ref`** stored in the "directory" field of the negotiation message indicates the directory that the file should be created in.

- "**name**": If the sender decides to (or is forced to) pass data to the receiver via a file, the string in this field should be used as the name of the file. **Note**: the receiver should create the file before deciding on a file name, to ensure that file name is in fact available, and will remain available.

Of course, you can leave out fields that don't apply to a particular action. For example, if the receiver chooses **`B_TRASH_TARGET`** as the action (by putting a value of **`B_TRASH_TARGET`** into the **`what`** member of the negotiation message), no message fields are required; in fact, the sender doesn't even need to respond to a request of **`B_TRASH_TARGET`** with a data message, it simply has to delete the dragged data.

Protocol for the Data Message

The data message is the third and final message sent as part of a negotiated drag'n'drop. It is sent in response to a negotiation message, using the **`BMessage::SendReply()`** function, and is sent only if the sender application has elected to pass the dragged

data directly in a message. If the sender application passes data to the receiver via a file, no data message is sent. [*xxx just want to check, is this correct? It would seem you'd want to at least send confirmation that the write was successful. Or should the receiver just do a node monitor kind of thing?]*

When Data Is Sent in the Data Message

If the sender elects to send the dragged data directly in the data message, then the data message will be constructed with the following structure:

- A **`what`** member set to the value of the **`B_MIME_DATA`** constant.
- A single message field, whose name indicates the type of mime data being sent, and whose contents consist of the data itself. For example, if you drag some text from one text processing window to another, the final data message (after the drop and negotiation messages have been sent) might contain a message field named "text/plain", which contains the text being dragged. Alternatively, if you dragged text from an HTML viewer, the data message might (depending on the negotiations) have a single field called "text/html", which would contain the text *including* the html formatting statements.

Note that when the data messages arrives at the receiver, it may contain more than the single message field described above; other message fields might be added automatically, by parts of the system. However, the message field above is the only one *you* add.

Handling Drops from 'Old' Applications

Negotiated drag'n'drop was defined relatively recently; older applications may still use the "old-style" drag'n'drop. It's simple for your application to take into the account that it might be a receiver for such a drop.

Under the old style drag'n'drop, a single message was sent from the sender to the receiver, with a what member of **`B_MIME_DATA`**, and the value of the dragged data in the "be:data" message field. *[xxx how is the type of the enclosed data determined?]* Since your receiver application will need to monitor for **`BMessages`** with a what field of **`B_MIME_DATA`** (i.e. your application will need to look for the data messages that might occur as part of a negotiated drag'n'drop), it's simple to add in a bit more code that handles the fact that such a message may occur without being part of a negotiation, and may contain a "be:data" field.

Summary of the Negotiated Drag'n'Drop Protocol

Drag Message

The structure of a drag message is as follows:

- A **`what`** member of **`B_SIMPLE_DATA`**
- Three message fields "**be:types**", "**be:filetypes**" and "**be:type_descriptions**", which contain (respectively) formats the sender is willing to pass data in via the data message, formats the sender is willing to pass data via a file, and user-friendly descriptions of the different formats.
- A message field "**be:actions**" containing a list of actions the sender is willing to perform at the request of the receiver.
- An optional field "**be:clip_name**" indicating a suggested name for a clipping.
- Optional fields "**be:originator**" and "**be:originator_data**", which may be used to pass store context information during an asynchronous drag'n'drop.
- "**_drop_point_**" and "**_drop_offset_**", which are filled in by the system, and provide information about where the mouse cursor was positioned when the drop took place.

Negotiation Message

The structure of a negotiation message is as follows:

- A **`what`** member, which is one of the values extracted from the "be:actions" field of the drag message to which this negotiation message is a reply.
- A list of formats in the "**be:types**" message field, chosen from the list provided in the "be:types" of the drag message. This indicates the formats the receiver is willing to accept data in. If the value given by the global constant **`B_FILE_MIME_TYPE`** is in the list, then the receiver is willing to accept data through a file, as well as through the data message. Note that values after an entry of **`B_FILE_MIME_TYPE`** will be ignored; thus, if **`B_FILE_MIME_TYPE`** is the first element of "be:types", then the receiver is willing to accept data *only* through a file. If there is more than one value in "be:types", the first value is the preferred format.
- A list of formats in the "**be:filetypes**" message field, which indicates file formats the receiver is willing to accept, if data is passed in a file. This is meaningful only if "be:types" contains a value of **`B_FILE_MIME_TYPE`** as one of its values. If there is more than one value in "be:filetypes", the first value is the preferred format.
- "**directory**" and "**name**" fields, which should be provided if the negotiation message has stated that the receiver is willing to accept data via a file. "directory" is an entry_ref specifying where the file should be placed, and "name" is a string specifying the desired name of the file.

Data Message

The structure of a data message is as follows:

- A **`what`** member of **`B_MIME_DATA`**.
- A single **data message field** added by the program code; the *name* of this field is the mime type of the enclosed data, and the *value* of this field is the data itself. For example, a block of plain text would be sent in a message field named "text/plain". The mime type for the enclosed data is chosen from the list of formats in the "be:types" field of the negotiation message to which this message is a reply.

Be Book Table of Contents

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved.

Text last modified January 26, 2000.

Tracker Table of Contents

Tracker Scripting
Tracker Add-on Protocol

The Be Book Table of Contents Tracker Index

The Be Book,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved.

The Be Book Tracker

Tracker: Master Index

E

The Entry Property	Tracker Scripting

P

process_refs()	Tracker Add-on Protocol

S

The Selection Property	Tracker Scripting

T

Tracker Add-on Protocol	Tracker Add-on Protocol
Tracker Add-on Protocol	Tracker Add-on Protocol
Tracker Scripting	Tracker Scripting
Tracker Scripting	Tracker Scripting

The Be Book **Tracker**

,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Tracker Scripting

Each Tracker window defines a "Poses" property representing the contents of the window. Each poses, in turn, defines the two properties "Entry" and "Selection". An "Entry" is an item in the window, e.g. either a file or a directory, while a "Selection" represents a selected "Entry".

When a Tracker window receives a scripting message with a "Poses" property, it pops the current specifier off the specifier stack and then forwards the scripting message to the view handling the "Poses" property. From there, the "Entry" and "Selection" properties are processed. For example, the following function returns the number of entries present in a given Tracker window:

```
int32 CountEntries(const char *name)
{
int32 count;
BMessage message, reply;

// form scripting request
message.what = B_COUNT_PROPERTIES;
message.AddSpecifier("Entry");
message.AddSpecifier("Poses");
message.AddSpecifier("Window", name);

// deliver request and fetch response
BMessenger("application/x-vnd.Be-TRAK").SendMessage(&message, &reply);

// return result
if (reply.FindInt32("result", &count) == B_OK)
return count;

return -1;
}
```

The Tracker scripting API defines a number of ways of specifying entries in a Poses. These methods are summarized below:

Specifier	Description
B_DIRECT_SPECIFIER	Used for specifying the entire Poses or selection as appropriate.
B_INDEX_SPECIFIER	"index" contains int32 index of file in the Poses. Ranges are specified with a pair of indices.
'sref'	"refs" contains entry_refs of specified files.
'sprv'	Refers to item immediately following file whose entry_ref is found in "data."
'snxt'	Refers to item immediately preceeding file whose entry_ref is found in "data."

Always remember that other programs (or the user) may also be adding or removing entries to the view and selection, so do not rely upon indices as a safe method of referring to a specific file. Instead, use entry_refs.

The Entry Property

Message	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Counts entries in a Poses.
B_DELETE_PROPERTY	'sref', B_INDEX_SPECIFIER	Moves the specified entry to the Trash.
B_EXECUTE_PROPERTY	'sref', B_INDEX_SPECIFIER	Perform the equivalent action of opening the specified items in the Tracker.
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns entry_refs of all entries in current Poses.
B_GET_PROPERTY	B_INDEX_SPECIFIER	Returns specified entry_ref.
B_GET_PROPERTY	'sprv', 'snxt'	Returns entry_ref of entry prior to or following specified entry_ref. Also returns index of file in "index."

The Selection Property

Message	Specifiers	Meaning
B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Counts the number of selected items in the Poses.
B_CREATE_PROPERTY	B_DIRECT_SPECIFIER	Adds items to the current selection. These can be specified as either entry_refs or int32s in the "data" array.
B_DELETE_PROPERTY	'sref', B_INDEX_SPECIFIER	Removes items from the current selection.
B_GET_PROPERTY	B_DIRECT_SPECIFIER	Returns entry_refs of items in selection.
B_GET_PROPERTY	'sprv', 'snxt'	Returns entry_ref of file prior to or following given item. Returns the index of the file in "index."
B_SET_PROPERTY	B_DIRECT_SPECIFIER	Clears the current selection and set it to the range given in "data." Also accepts entry_refs in "data" to determined the new selection.
B_SET_PROPERTY	'sprv', 'snxt'	Clears the current selection and sets it to the entry_refs prior to or following those specified in "data."

The Be Book
,
...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..

Tracker Add-on Protocol

The Tracker provides a convenient shortcut mechanism through the use of add-ons. A user can access a special add-ons menu by right clicking in the Tracker. The Tracker interacts with an add-on through the **process_refs()** function described below.

Tracker add-ons should be placed in **/boot/home/config/add-ons/Tracker**. A shortcut key can be associated with the add-on by appending a dash followed by the shortcut key to the filename of the add-on.

---

process_refs()

Declared in: be/**be/add-ons/tracker/TrackerAddOn.h**

void **process_refs(**entry_ref *dir_ref*, BMessage **msg*, void **reserved***)*

The Tracker calls this function when the user invokes the add-on. The current directory is found in *dir_ref*. *msg* is a standard **B_REFS_RECEIVED** BMessage with the "*refs*" array containing the **entry_ref**s of the files selected by the user. The third argument is currently unused.

`process_refs()` runs in a separate thread within the Tracker's team, so if your add-on crashes, the Tracker goes too.

A simple Tracker Add-On follows. It simply takes the contents of the arguments to `process_refs()` and outputs them in a window.

```
#include <Application.h>
#include <InterfaceKit.h>
#include <StorageKit.h>

#include <stdio.h>
#include <string.h>

#include <be/add-ons/tracker/TrackerAddon.h>

void process_refs(entry_ref dir_ref, BMessage *msg, void *)
{
BWindow *window = new BWindow(BRect(100,100,300,300),
   "Sample Tracker Add-on", B_TITLED_WINDOW, 0);
BTextView *view = new BTextView(BRect(0,0,200,200), "view",
   BRect(0,0,200,200), B_FOLLOW_ALL_SIDES, B_WILL_DRAW |
   B_FULL_UPDATE_ON_RESIZE);

BPath path;
BEntry entry(&dir_ref);
entry.GetPath(&path);
```

```
view->Insert("Current Directory: ");
view->Insert(path.Path());
view->Insert("n");

int refs;
entry_ref file_ref;
for (refs=0;msg->FindRef("refs", refs, &file_ref) == B_NO_ERROR;refs++) {
if (refs == 0)
      view->Insert("Selected files:n");
entry.SetTo(&file_ref);
entry.GetPath(&path);
view->Insert(path.Path());
view->Insert("n");
}

if (refs == 0)
view->Insert("No files selected.n");

view->MakeEditable(false);
window->AddChild(view);
window->Show();
}

main()
{
new BApplication("application/x-sample-tracker-add-on");
(new BAlert("", "Sample Tracker Add-on", "swell"))->Go();
delete be_app;
}
```

The Be Book Tracker Tracker Index

The Be Book

,

...in lovely HTML...
for BeOS Release 5.

Copyright © 2000 Be, Inc. All rights reserved..